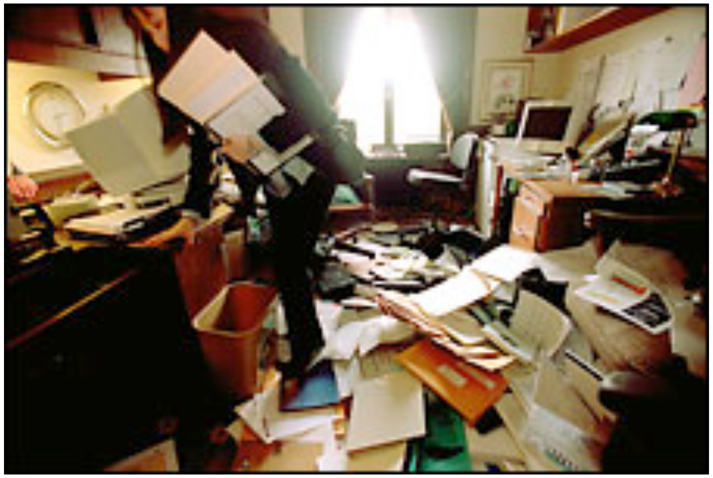
National Energy Research Scientific Computing Center (NERSC)

Observations on I/O Requirements for HPC Applications: *A User Perspective*

John Shalf
NERSC Center Division, LBNL

DARPA Exascale Meeting
September 6, 2007

Motivation and Problem Statement



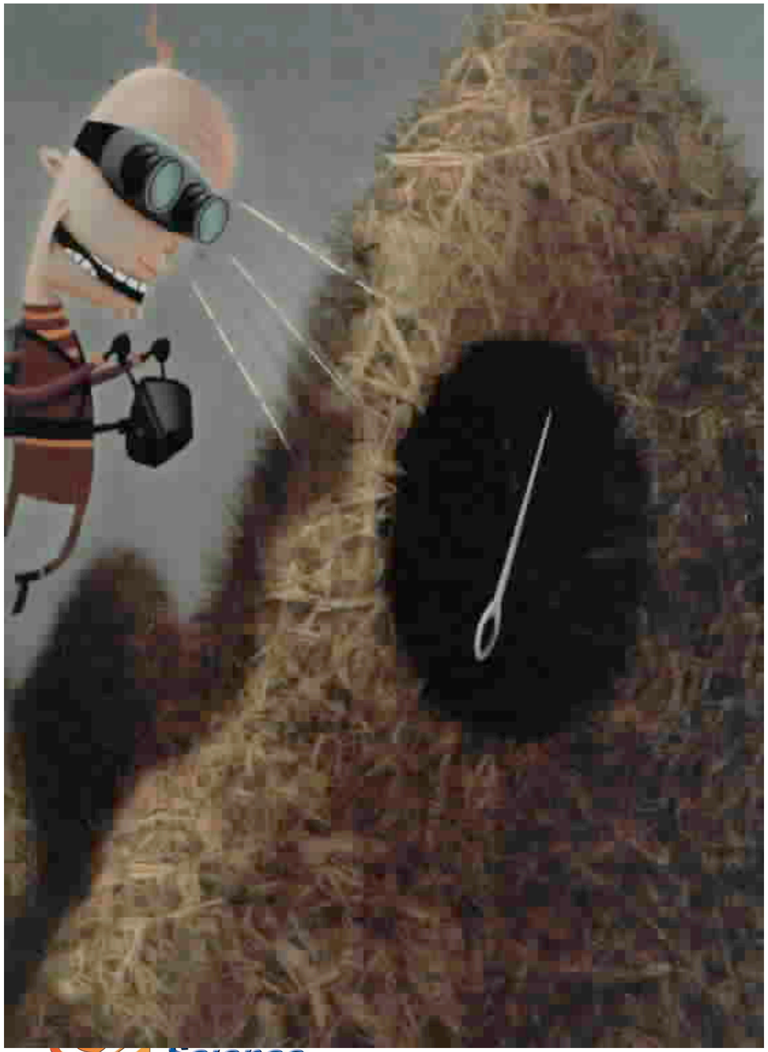
- Too much data.
- Data Analysis "meat grinders" not especially responsive to needs of scientific research community.
- What scientific users want:
  - Scientific Insight
  - Quantitative results
  - Feature detection, tracking, characterization
  - (lots of bullets here omitted)
- See:

  http://vis.lbl.gov/Publications/2002/VisGreenFindings-LBNL-51699.pdf

  http://www-user.slac.stanford.edu/rmount/dm-workshop-04/Final-report.pdf

Wes Bethel

Motivation and Problem Statement



- Too much data.
- Analysis "meat grinders" not especially responsive to needs of scientific research community.
- What scientific users want:
  - Scientific Insight
  - Quantitative results
  - Feature detection, tracking, characterization
  - (lots of bullets here omitted)
- See:

  http://vis.lbl.gov/Publications/2002/VisGreenFindings-LBNL-51699.pdf

  http://www-user.slac.stanford.edu/rmount/dm-workshop-04/Final-report.pdf

Parallel I/O: *A User Perspective*

- **Requirements (desires)**
  - Write data from multiple processors into a single file
  - Undo the “domain decomposition” required to implement parallelism
  - File can be read in the same manner regardless of the number of CPUs that read from or write to the file. (eg. we want to see the logical data layout… not the physical layout)
  - Do so with the same performance as writing one-file-per-processor (only writing one-file-per-processor because of performance problems)

  seems simple: but scientists are tough customers
- **Scientists and Application Developers**
  - *Cannot agree on anything (Always roll their own implementation)*
  - *Only care about their OWN data model and requirements*
  - *Cannot tell the difference between a file format and a data schema (so they end up being one-in-the-same)*
  - *Are forced to specify physical layout on disk by existing APIs*
    - *Always make the wrong choices when forced to do so!*
    - *Always blame the filesystem or hardware when the performance is terrible*

Parallel I/O: *A User Perspective*

- **Requirements (desires)**
  - Write data from multiple processors into a single file
  - Undo the "domain decomposition" required to implement parallelism
  - File can be read in the same manner regardless of the number of CPUs that read from or write to the file. (eg. we want to see the logical data layout… not the physical layout)
  - Do so with the same performance as writing one-file-per-processor (only writing one-file-per-processor because of performance problems)

  seems simple: but scientists are tough customers
- **Scientists and Application Developers**
  - *Cannot agree on anything (Always roll their own implementation)*
  - *Only care about their OWN data model and requirements (forget IGUDM)*
  - *Cannot tell the difference between a file format and a data schema (so they end up being one-in-the-same)*
  - *Are forced to specify physical layout on disk by existing APIs*
    - *Always make the wrong choices when forced to do so!*
    - *Always blame the filesystem or hardware when the performance is terrible*
  - ***I have spent most of my career as one of those people!***

Usage Model

- **Checkpoint/Restart**
  - Typically not functional until ~1 month before the system is retired
  - Length of time between system introduction and functional CPR growing
  - Most users don’t do hero applications: tolerate failure by submitting more jobs (and that includes apps that are targetting hero-scale applications)
  - Most people doing “hero applications” have written their own restart systems and file formats
  - Typically close to memory footprint of code per dump
    - Must dump memory image ASAP!
    - Not as much need to remove the domain decomposition (recombiners for MxN problem)
    - not very sophisticated about recalculating derived quantities (stores all large arrays)
    - Might go back more than one checkpoint, but only need 1-2 of them online (staging)
    - Typically throw the data away if CPR not required
- **Data Analysis Dumps**
  - Time-series data most demanding
    - Typically run with coarse-grained time dumps
    - If something interesting happens, resubmit job with higher output rate (and take a huge penalty for I/O rates)
    - FLASH code: select output rate to do < 10% of exec time… full dump costs 30% or more (up to 60% of exec time) (info from Katie Antypas)
    - Async I/O would make 50% I/O load go away, but nobody uses it! (rarely works)
  - Optimization or boundary-value problems typically have flexible output requirements (typically diagnostic)

Finding Data

- **Use clever file names to indicate data contents**
- **Use extensions to indicate format**
  - However, subtle changes in file format can render file unreadable
  - Mad search to find sub-revision of “reader” to read an older version of a file
  - Consequence of confusing file format with data model (common in this community)
- **Tend to get larger files when hierarchical self-describing formats are used**
  - Filesystem metadata (clever file names) replaced by file metadata
  - File as “object database container”
- Indexing
  - Metadata indices (SRMs, Metadata Catalogs)
  - Searching individual items within a dataset (FastBit)

Common Storage Formats

- **ASCII:** *(pitiful… this is still common… even for 3D I/O… and you want an exaflop??)*
  - Slow
  - Takes more space!
  - Inaccurate
- Binary
  - Nonportable (eg. byte ordering and types sizes)
  - Not future proof
  - Parallel I/O using MPI-IO
- Self-Describing formats
  - NetCDF/HDF4, HDF5, Silo
  - Example in HDF5: API implements Object DB model in portable file
  - Parallel I/O using: pHDF5/pNetCDF (hides MPI-IO)
- Community File Formats
  - FITS, HDF-EOS, SAF, PDB, Plot3D
  - Modern Implementations built on top of HDF, NetCDF, or other self-describing object-model API

Common Data Models/Schemas

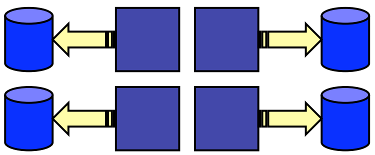
- **Structured Grids:**
  - 1D-6D domain decomposed mesh data
  - Reversing Domain Decomposition results in strided disk access pattern
  - Multiblock grids often stored in chunked fashion
- **Particle Data**
  - 1D lists of particle data (x,y,z location + physical properties of each particle)
  - Often non-uniform number of particles per processor
  - PIC often requires storage of Structured Grid together with cells
- **Unstructured Cell Data**
  - 1D array of cell types
  - 1D array of vertices *(x,y,z locations)*
  - 1D array of cell connectivity
  - Domain decomposition has similarity with particles, but must handle ghost cells
- **AMR Data (not too common yet)**
  - Chombo: Each 3D AMR grid occupies distinct section of 1D array on disk *(one array per AMR level).*
  - Enzo (Mike Norman, UCSD): One file per processor *(each file contains multiple grids)*
  - BoxLib: One file per grid *(each grid in the AMR hierarchy is stored in a separate,cleverly named, file)*
- **Increased need for processing data from terrestrial sensors (read-oriented)**
  - NERSC is now a net importer of data

Confusion about Data Models

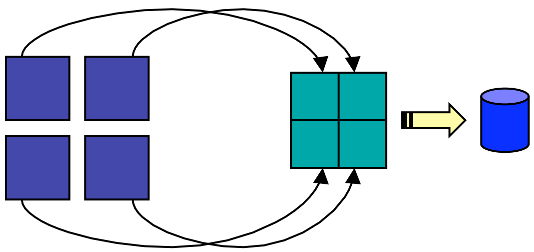
- **Scientist/App Developers generally confused about difference between Data Model and File Format**
  - Should use modern hierarchical storage APIs such as HDF5 or NetCDF
  - Performance deficiencies in HDF5 and pNetCDF generally traced back to performance of Underlying MPI-IO layer
    - Point to deficiency of forcing specification of physical layout
- **More Complex Data Models**
  - NetCDF is probably too weak of a data model
  - HDF5 is essentially an object database with portable self-describing file format
  - Fiber bundles is probably going TOO FAR

Common Physical Layouts

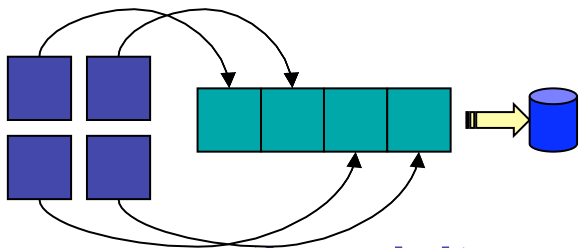
- One File Per Process
  - Terrible for HPSS!
  - Difficult to manage



- Parallel I/O into a single file
  - Raw MPI-IO
  - pHDF5 pNetCDF



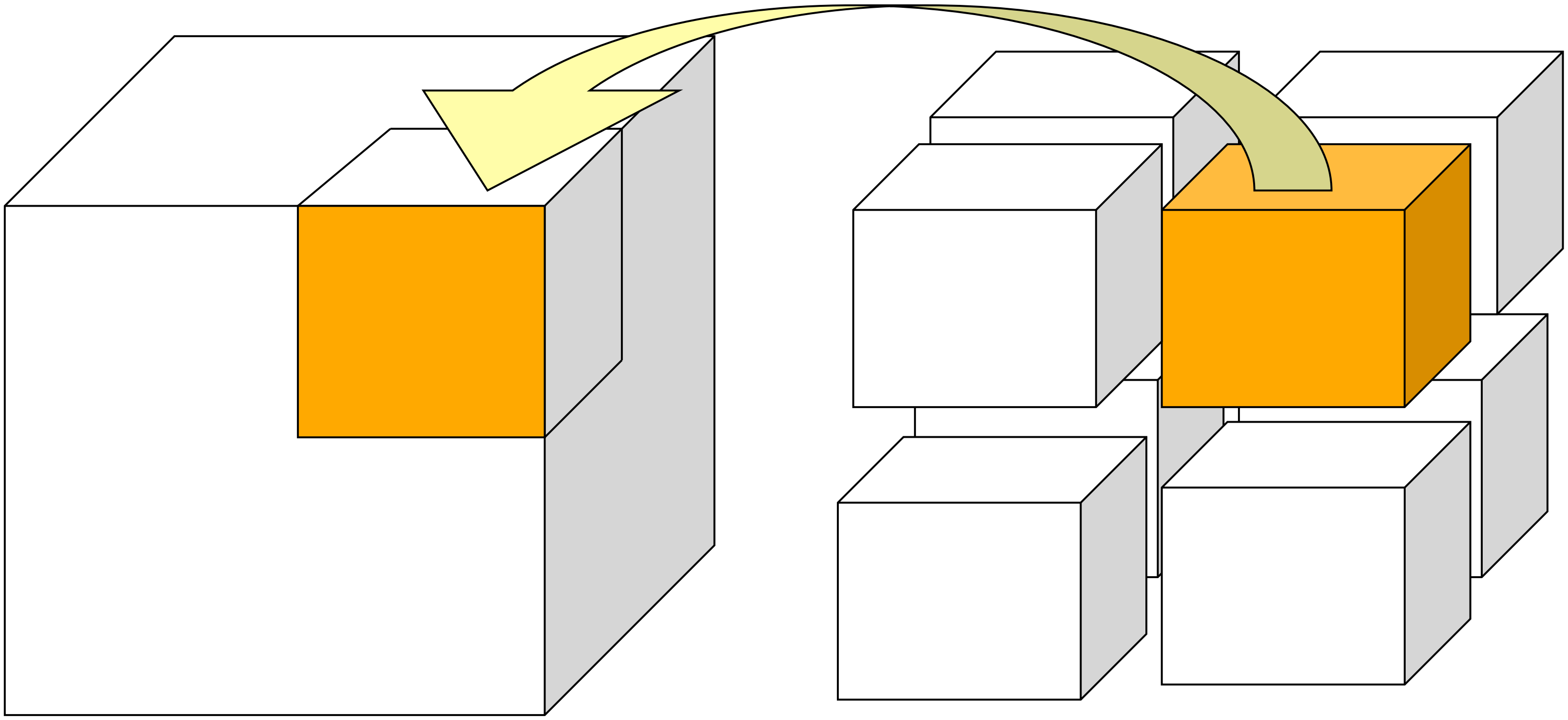
- Chunking into a single file
  - Saves cost of reorganizing data
  - Depend on API to hide physical layout
  - (eg. expose user to logically contiguous array even though it is stored physically as domain-decomposed chunks)



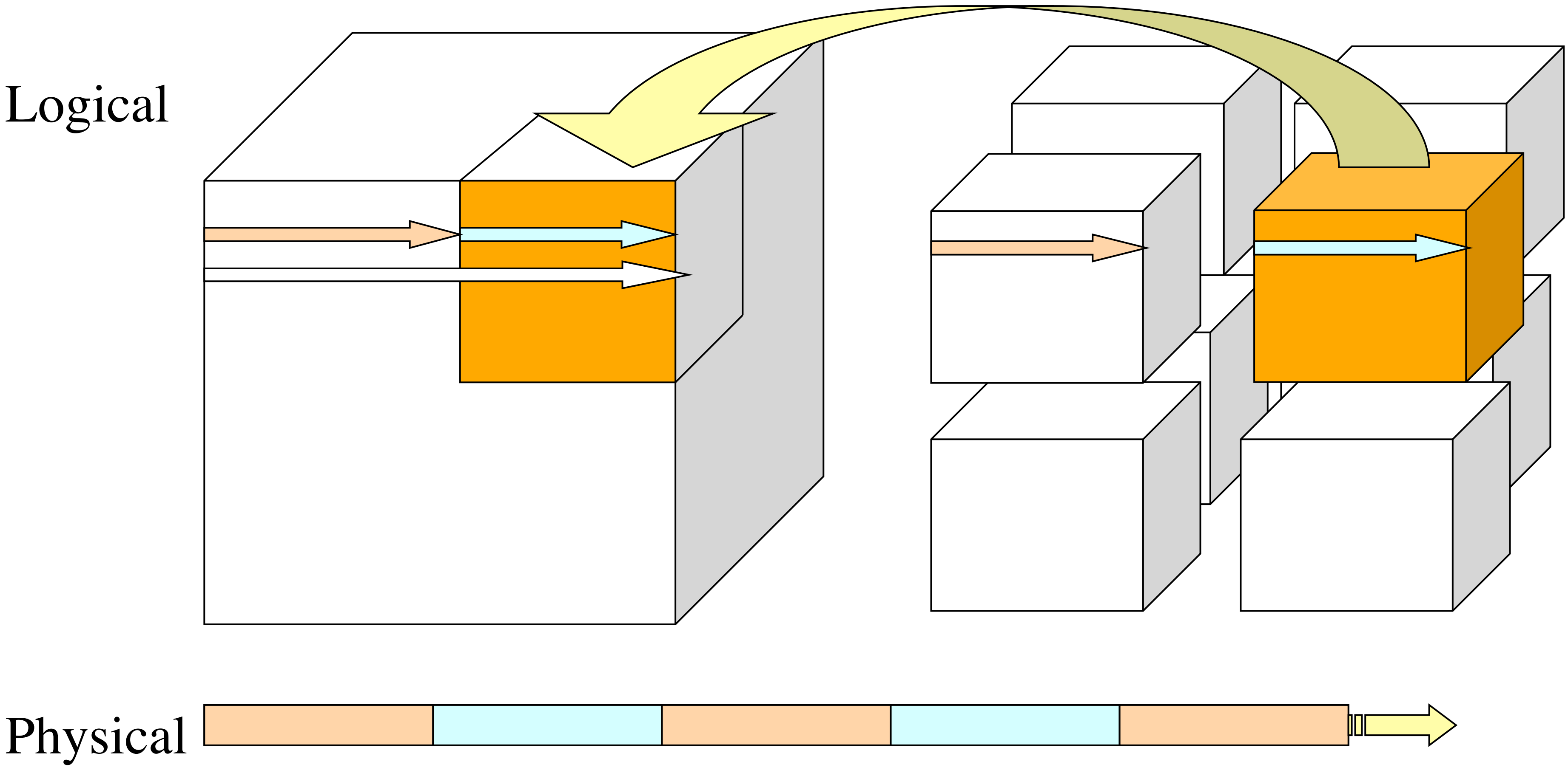
Common Themes for Storage Patterns

- Three patterns for parallel I/O into single file
  - >1D I/O: Each processor writes in a strided access pattern simultaneously to disk *(can be better organized… eg. PANDA)*
  - 1D I/O: Each processor writes to distinct subsections of 1D array *(or more than one array)*
  - 1D Irregular I/O: Each processor writes to distinct, but non-uniform subsections of 1D array (AMR, Unstructure Mesh Lists, PIC data)
- Three Storage Strategies
  - One file per processor (*terrible for HPSS!!!*)
  - One file per program: reverse domain decomp
  - One file per program: chunked output

3D (reversing the domain decomp)

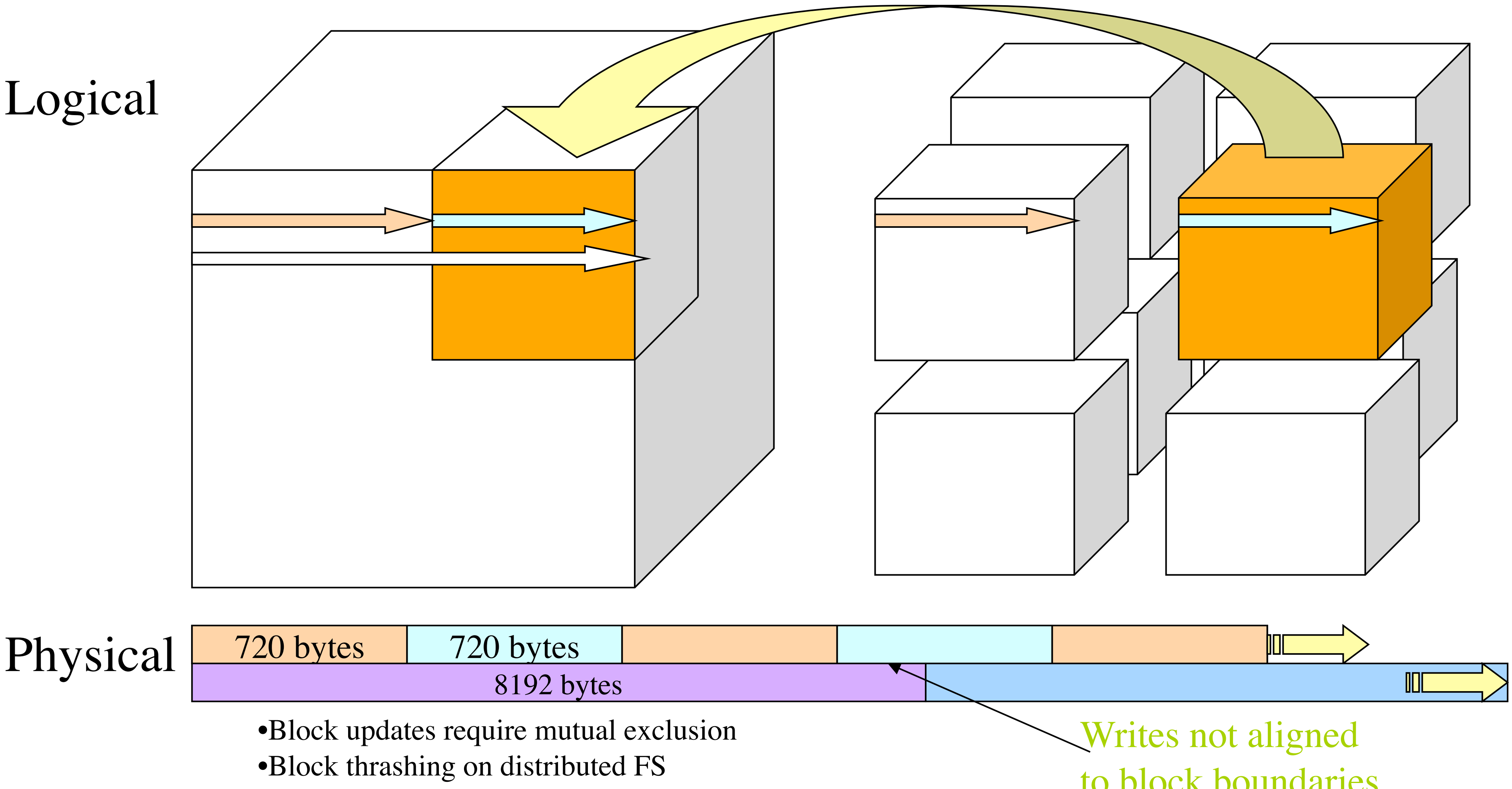


3D (reversing the decomp)



Office of Science
U.S. DEPARTMENT OF ENERGY

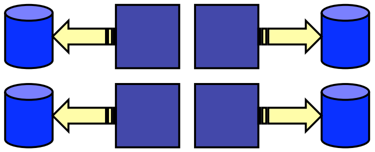
3D (block alignment issues)



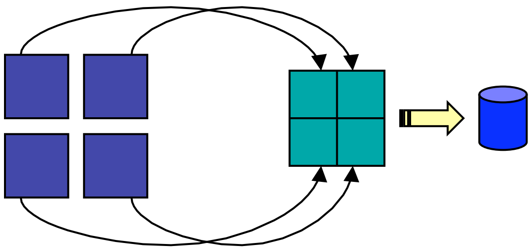
- Block updates require mutual exclusion
- Block thrashing on distributed FS
- I/O efficiency for sparse updates! (8k block required for 720 byte I/O operation
- Unaligned block accesses can kill performance! (but are necessary in practical I/O solutions)

Common Physical Layouts

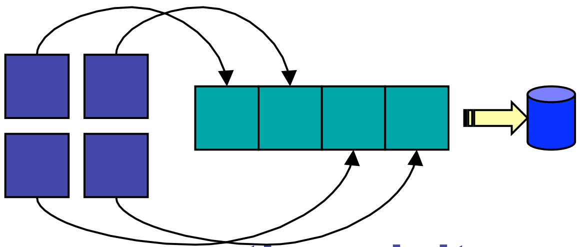
- One File Per Process
  - Terrible for HPSS!
  - Difficult to manage



- Parallel I/O into a single file
  - Raw MPI-IO
  - pHDF5 pNetCDF



- Chunking into a single file
  - Saves cost of reorganizing data
  - Depend on API to hide physical layout
  - (eg. expose user to logically contiguous array even though it is stored physically as domain-decomposed chunks)



Office of Science
U.S. DEPARTMENT OF ENERGY

Performance Experiences

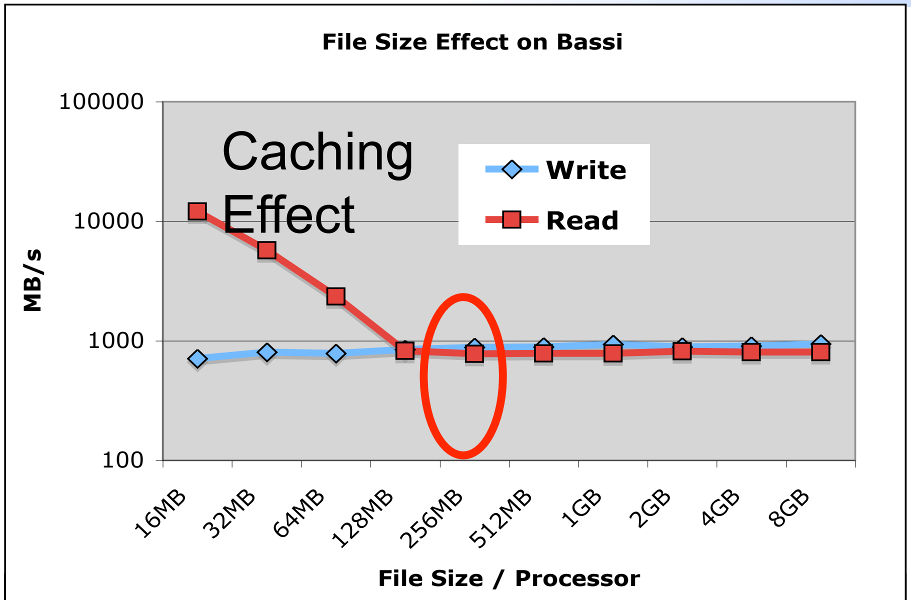
Platforms

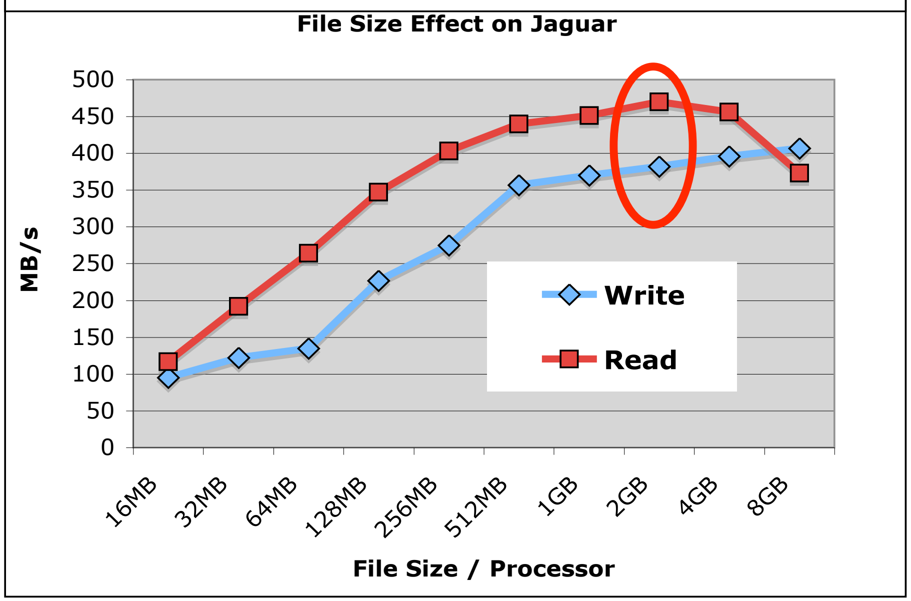
Machine Name	Parallel File System	Proc Arch	Inter-connect	Peak IO BW	Max Node BW to IO
Jaguar	Lustre	Opteron	SeaStar	18*2.3GB/s = 42GB	3.2GB/s (1.2GB/s)
Bassi	GPFS	Power5	Federation	6*1GB/s = ~6.0GB/s	4.0GB/s (1.6GB/s)

Effective unidirectional bandwidth in parenthesis

- 18 DDN 9550 couplets on Jaguar, each couplet delivers 2.3 - 3 GB/s
- Bassi has 6 VSDs with 8 non-redundant FC2 channels per VSD to achieve ~1GB/s per VSD. (2x redundancy of FC)

Caching Effects

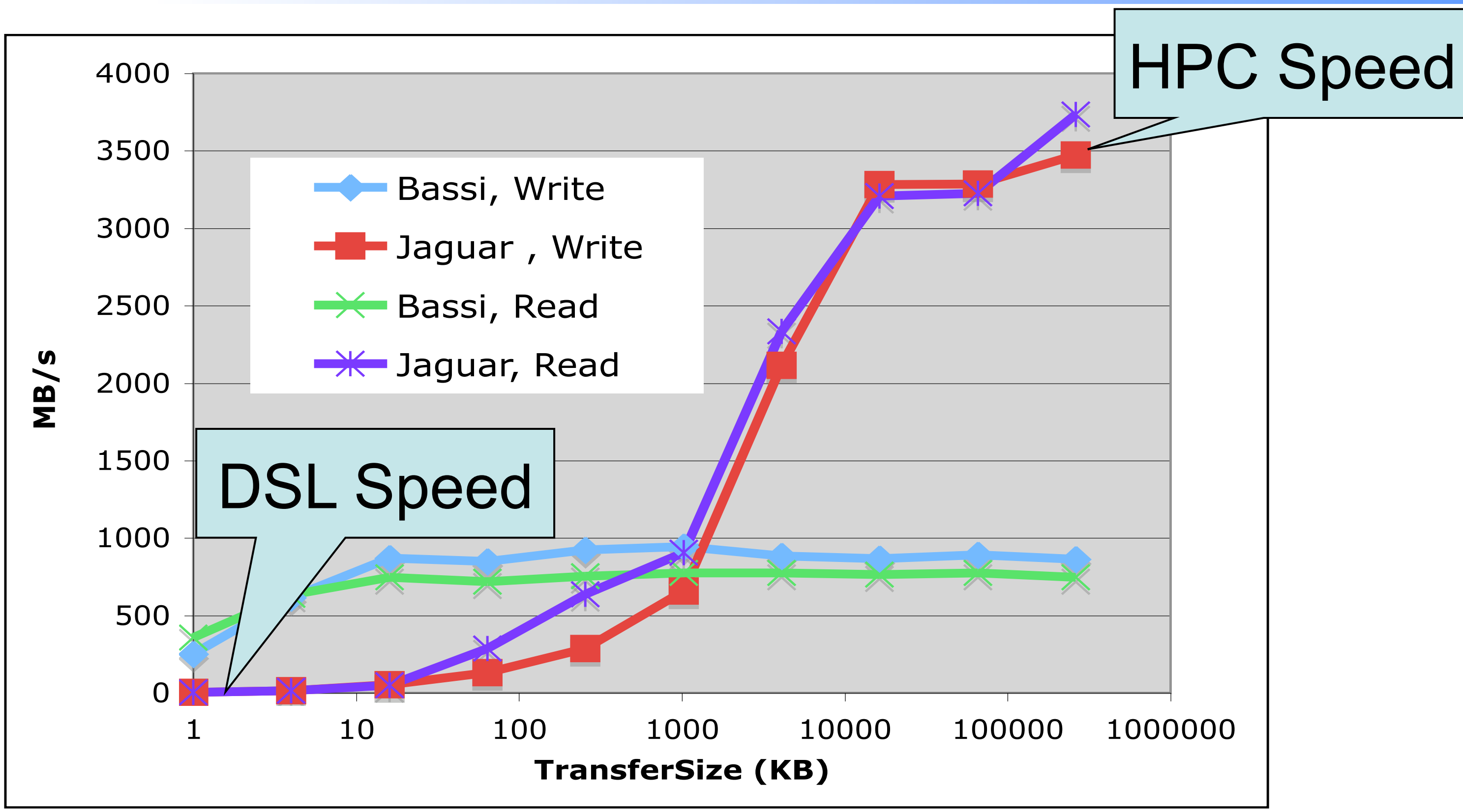




Machine Name	Mem Per Node	Node Size	Mem/ Proc
Jaguar	8GB	2	4GB
Bassi	32GB	8	4GB

- On Bassi, file Size should be at least 256MB/ proc to avoid caching effect
- On Jaguar, we have not observed caching effect, 2GB/s for stable output

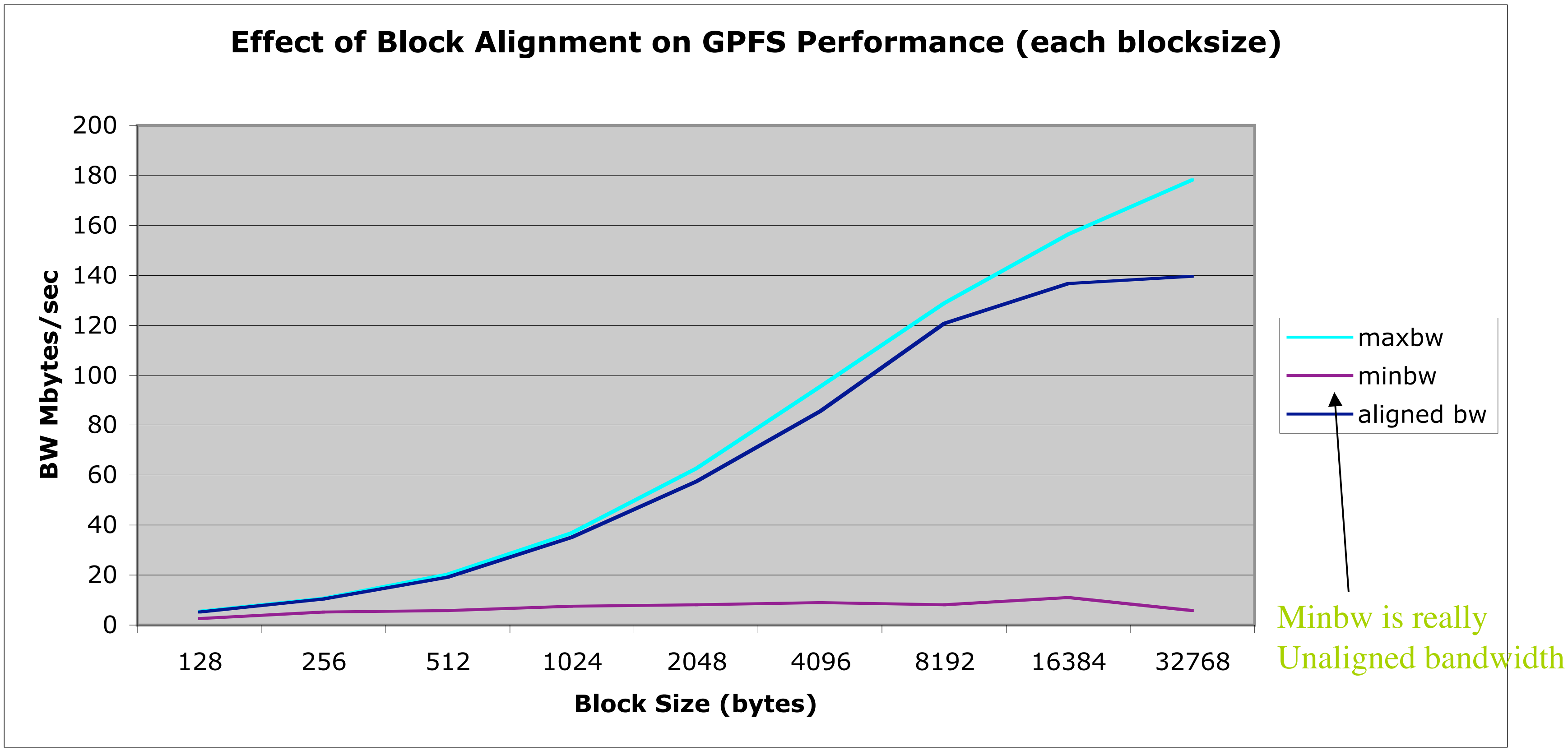
Transfer Size (P = 8)



- **Large transfer size is critical to achieve performance (common cause for weak perf.)**
- **Amdahl's law commonly kills I/O performance for small ops (eg. writing out record headers)**

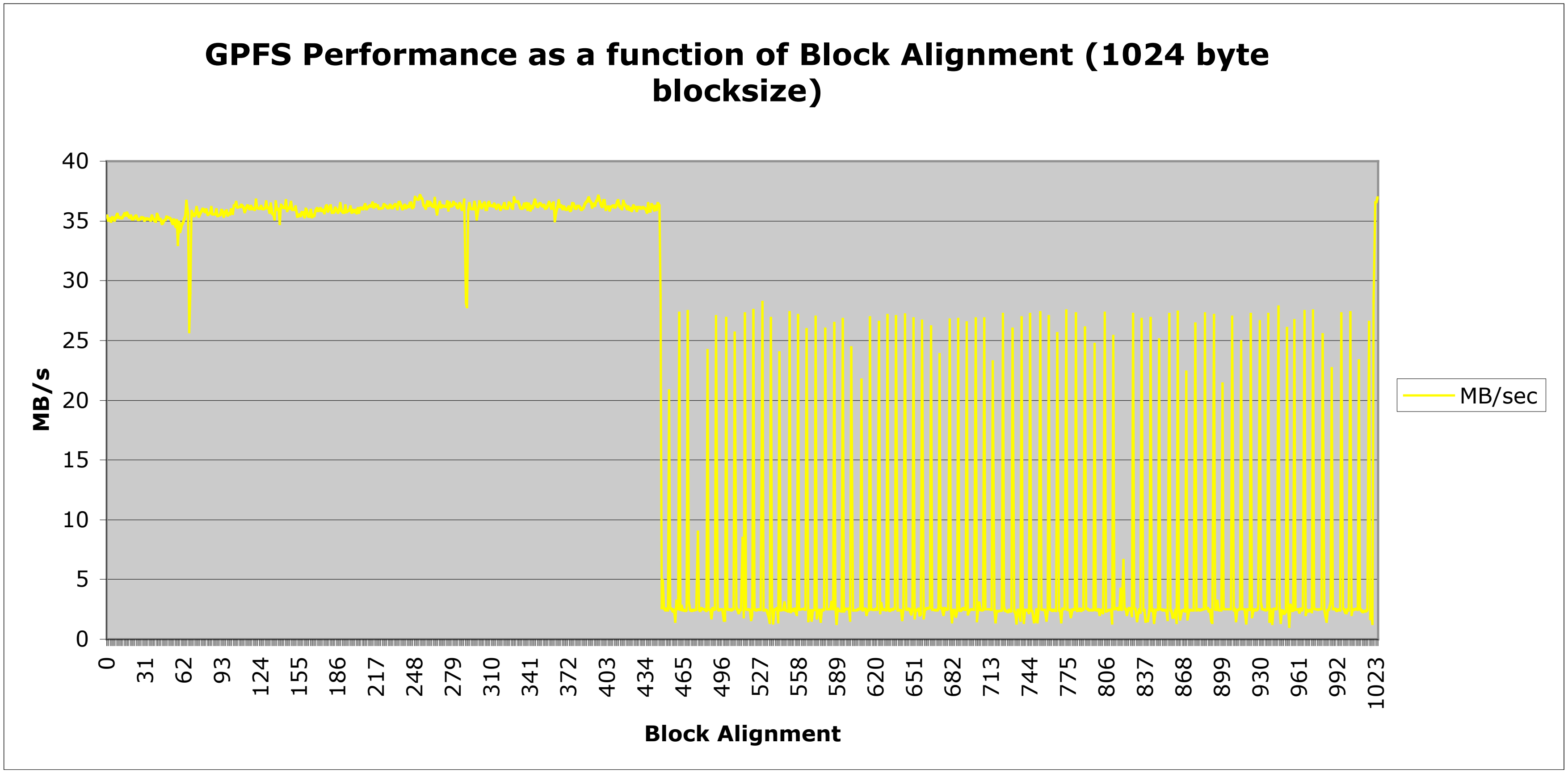
Office of Science
U.S. DEPARTMENT OF ENERGY

GPFS (unaligned accesses)

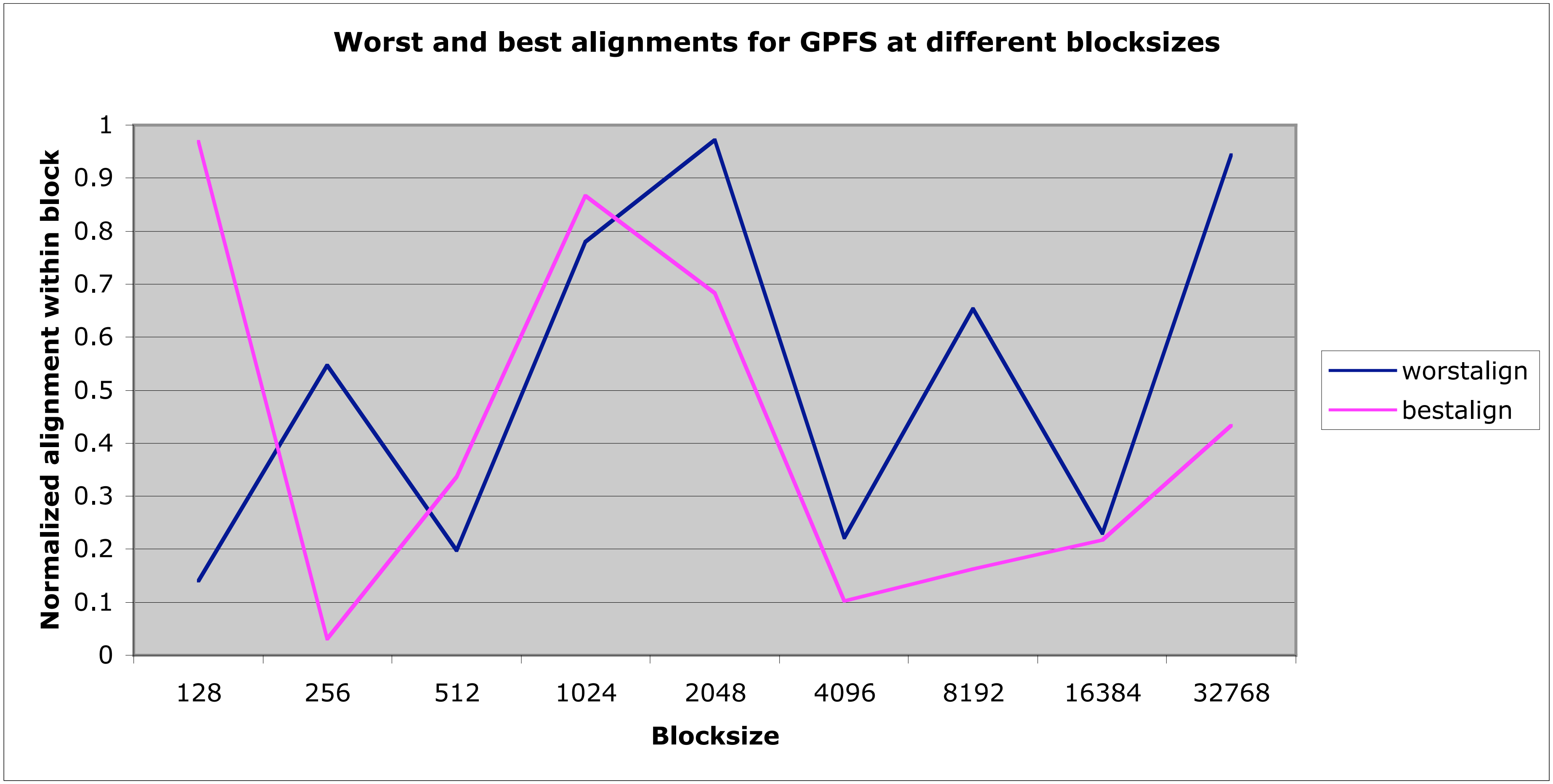


Unaligned access sucks!

GPFS: Unaligned accesses



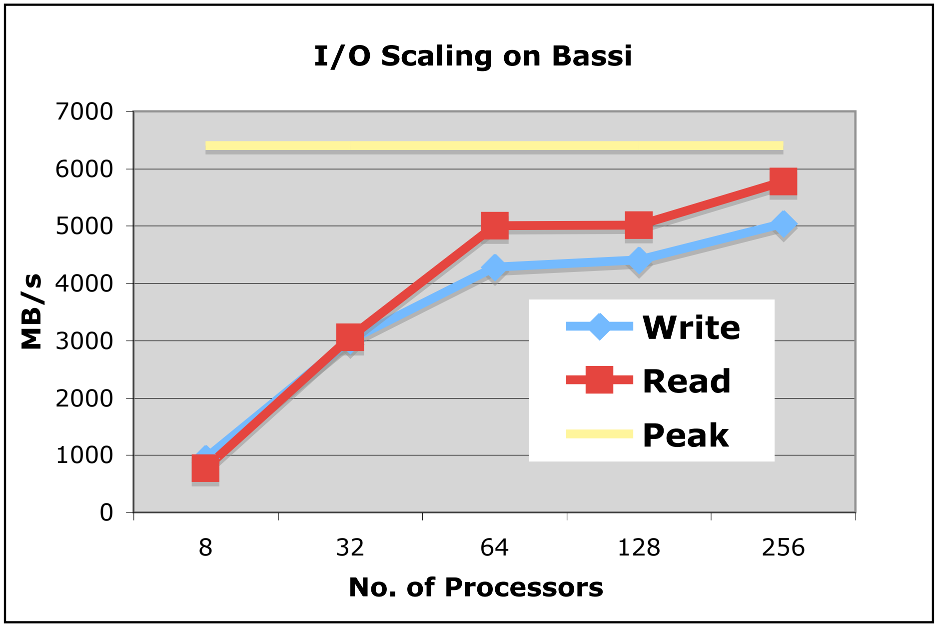
GPFS: (what alignment is best?)

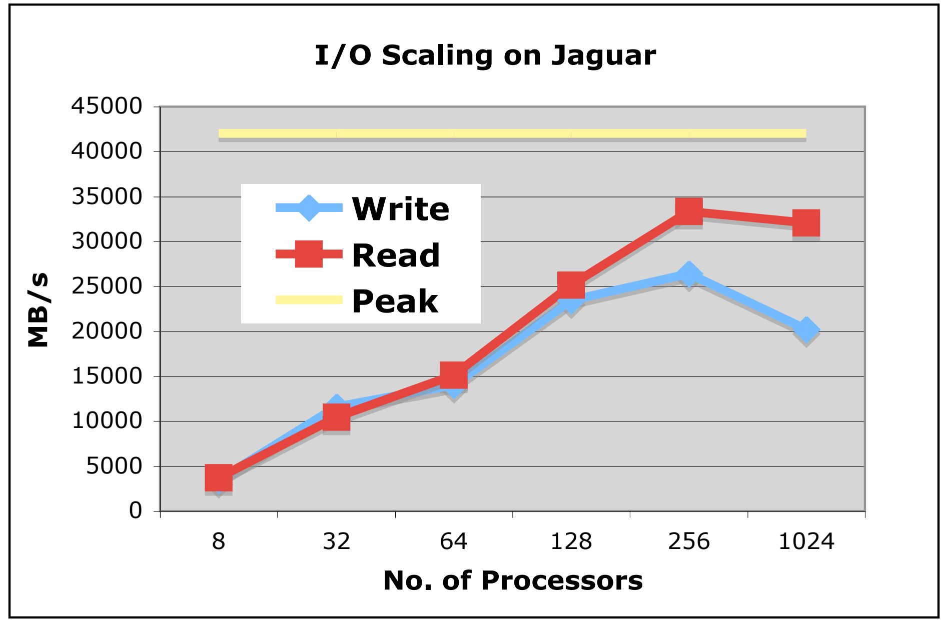


No consistently "best" alignment except for perfect block alignment!

That means 256k block boundaries for GPFS!

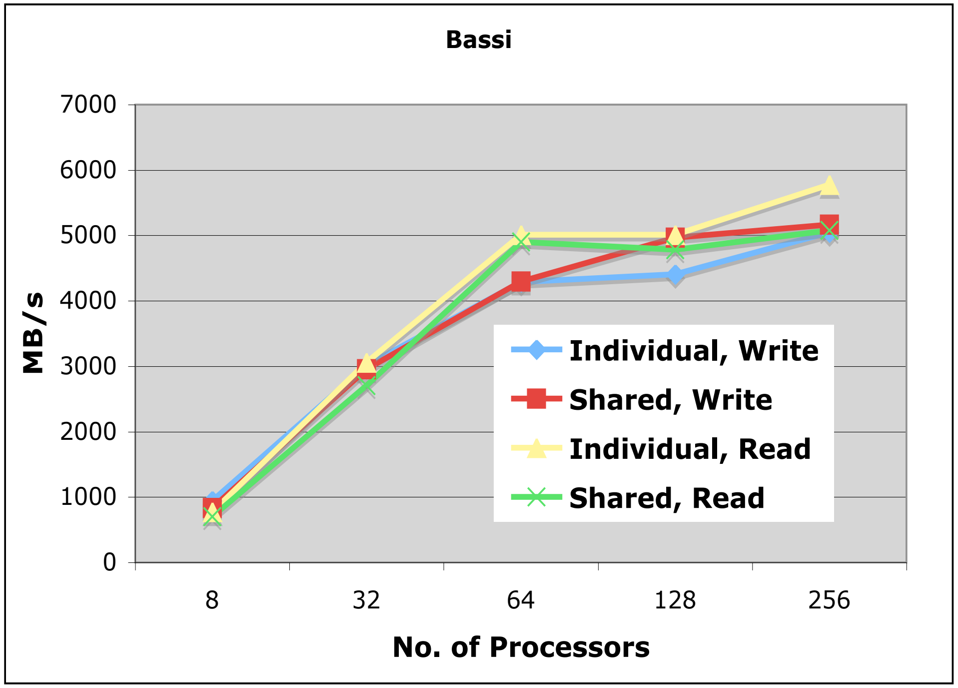
Scaling (No. of Processors)

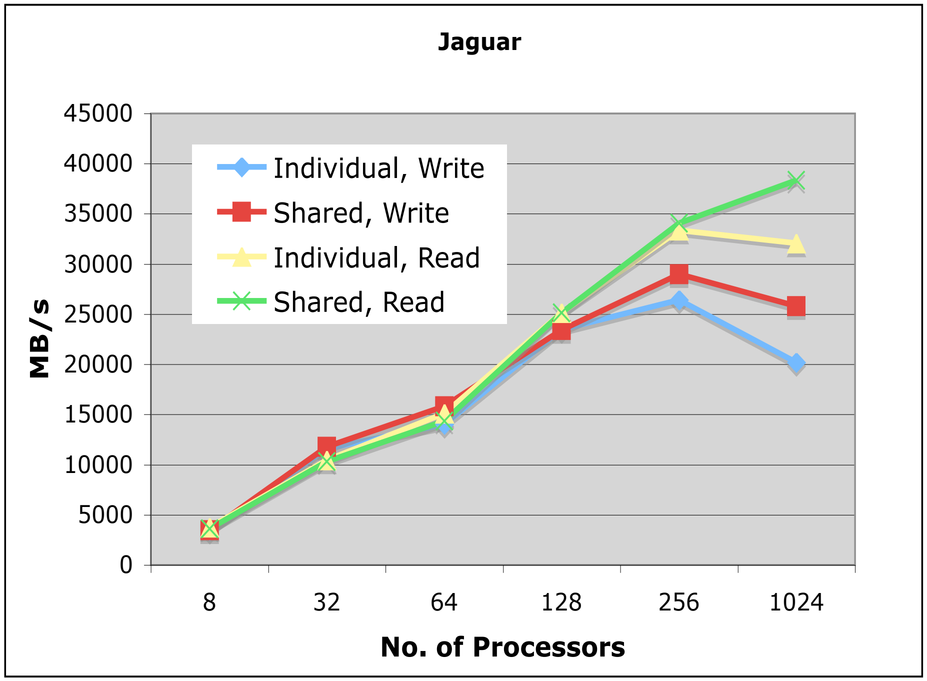




- The I/O performance peaks at:
  - P = 256 on Jaguar (lstripe=144),
  - Close to peaks at P = 64 on Bassi
- The peak of I/O performance can often be achieved at relatively low concurrency

Shared vs. One file Per Proc

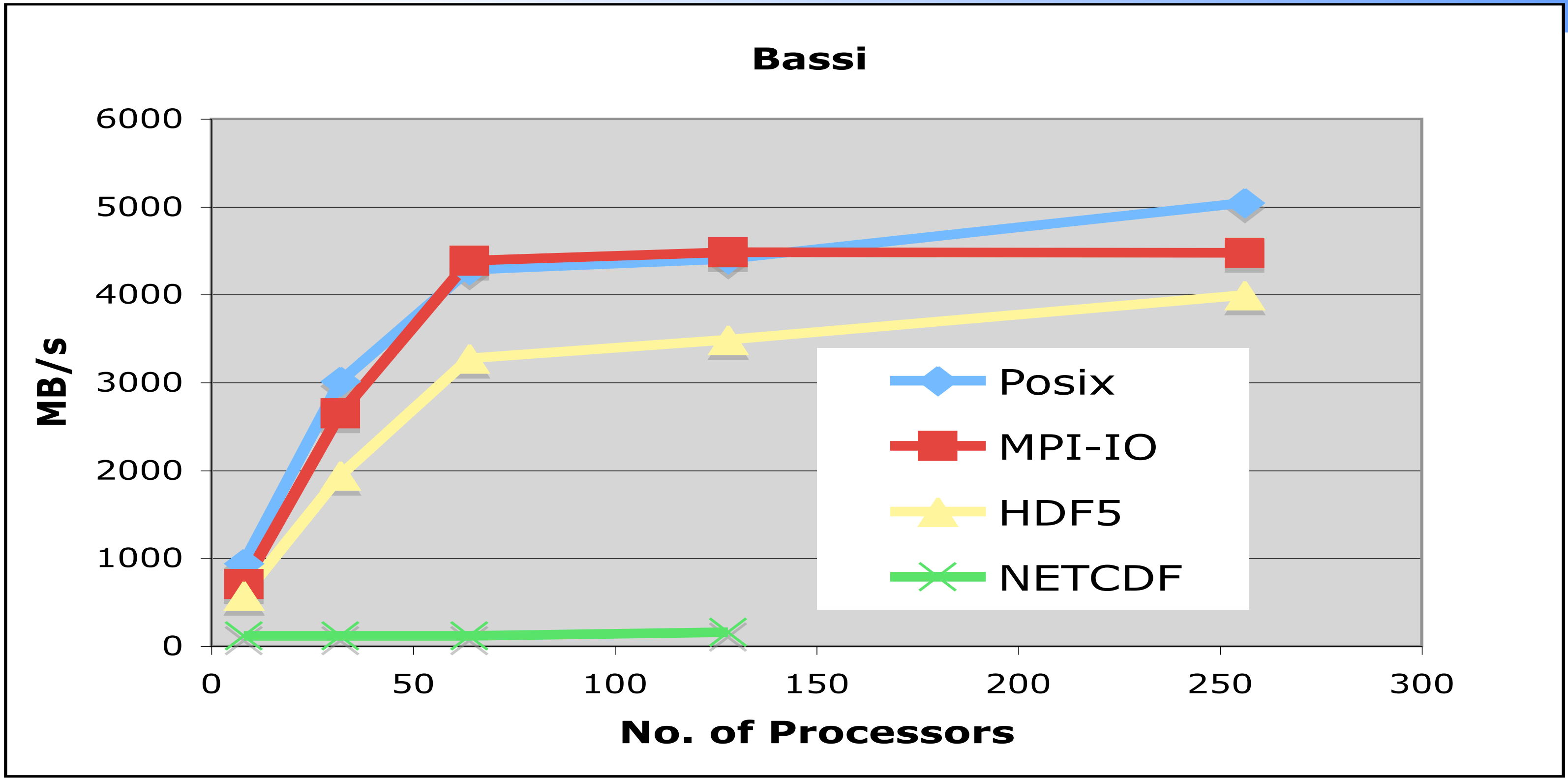




- The performance of using a shared file is very close to using one file per processor
- Using a shared file performs even better on Jaguar due to less metadata overhead

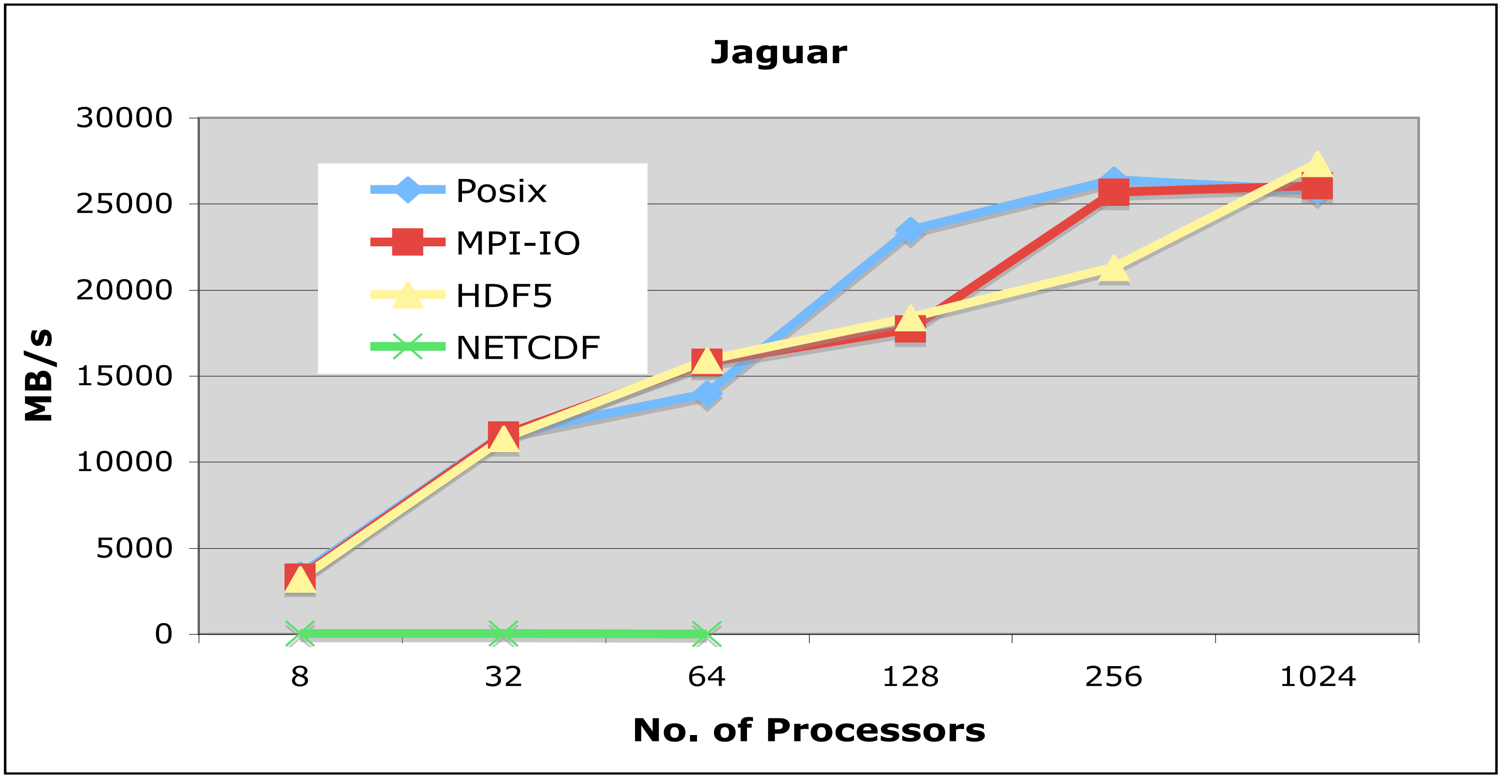
Office of Science
U.S. DEPARTMENT OF ENERGY

Programming Interface



- MPI-IO is close to POSIX performance
- Concurrent POSIX access to single-file works correctly
  - MPI-IO used to be required for correctness, but no longer
- HDF5 (v1.6.5) falls a little behind, but tracks MPI-IO performance
- parallelNETCDF (v1.0.2pre) performs worst, and still has 4GB dataset size limitation *(due to limits on per-dimension sizes on latest version)*

Programming Interface



- **POSIX, MPI-IO, HDF5 (v1.6.5) offer very similar scalable performance**
- **parallelNetCDF (v1.0.2.pre): flat performance**

Comments for DARPA

- **If you are looking at low-level disk access patterns, you are probably looking at the wrong thing**
  - Reflection of imperative programming interface that forces user to specify physical layout on disk
  - Users always make poor choices for physical layout
  - You will end up designing I/O for bad use case
- **Conclusion: Application developers forced to make bad choices by imperative APIs**
  - MPI-IO is a pretty good API for an imperative approach to describing mapping from memory to disk file layout
  - The imperative programming interface embodied by MPI-IO was the wrong choice! (we screwed up years ago and are paying the price now for our mistake!)
  - Lets not set new I/O system requirements based on existing physical disk access patterns -- consider logical data schema of the applications (more freedom for optimization)

Data Layout: Imperative vs. Declarative

- **Physical vs. Logical**
  - Physical Layout In Memory
  - Physical Layout on Disk
  - Logical Layout (data model): intent of application developer
- **Imperative Model**
  - Define physical layout in memory
  - Define physical intended physical layout on disk
  - Commit operation (read or write)
  - Performance
    - Limited by strict POSIX semantics (looking for "relaxed POSIX")
    - Compromised by Naïve users making wrong choices for phys layout
    - Limited freedom to optimize performance (data-shipping)
  - APIs: MPI-IO, POSIX
- **Declarative Model**
  - Define physical layout in memory
  - Define logical layout for "global view" of the data
  - Performance
    - Lower layers of the software get to make decisions about optimizing physical layout and annotate the file to record the choices that it made
    - User needn't be exposed to details of disk or "relaxed POSIX" semantics
  - API/Examples: PANDA. HDF5 has some elements, but limited by MPI-IO

Declarative vs. Imperative

- **Application developers really don't care (or shouldn't care) about physical layout**
  - Know physical layout in memory
  - Know desired "logical layout" for the global view of their "data"
  - Currently FORCED to define physical layout because the $%^&* API requires it!
  - When forced to define the physical (in memory) to physical (on disk) mapping, application developers always make the wrong choices!
  - Declarative model to specify desired logical layout would be better, and provide filesystems or APIs more freedom to optimize performance (e.g. Server Directed I/O)
- **DB Pioneers learned these lessons 50 years ago**
  - Our community is either stupid or arrogant for failing to heed these lessons (probably just arrogant)

Say something nice about server directed I/O

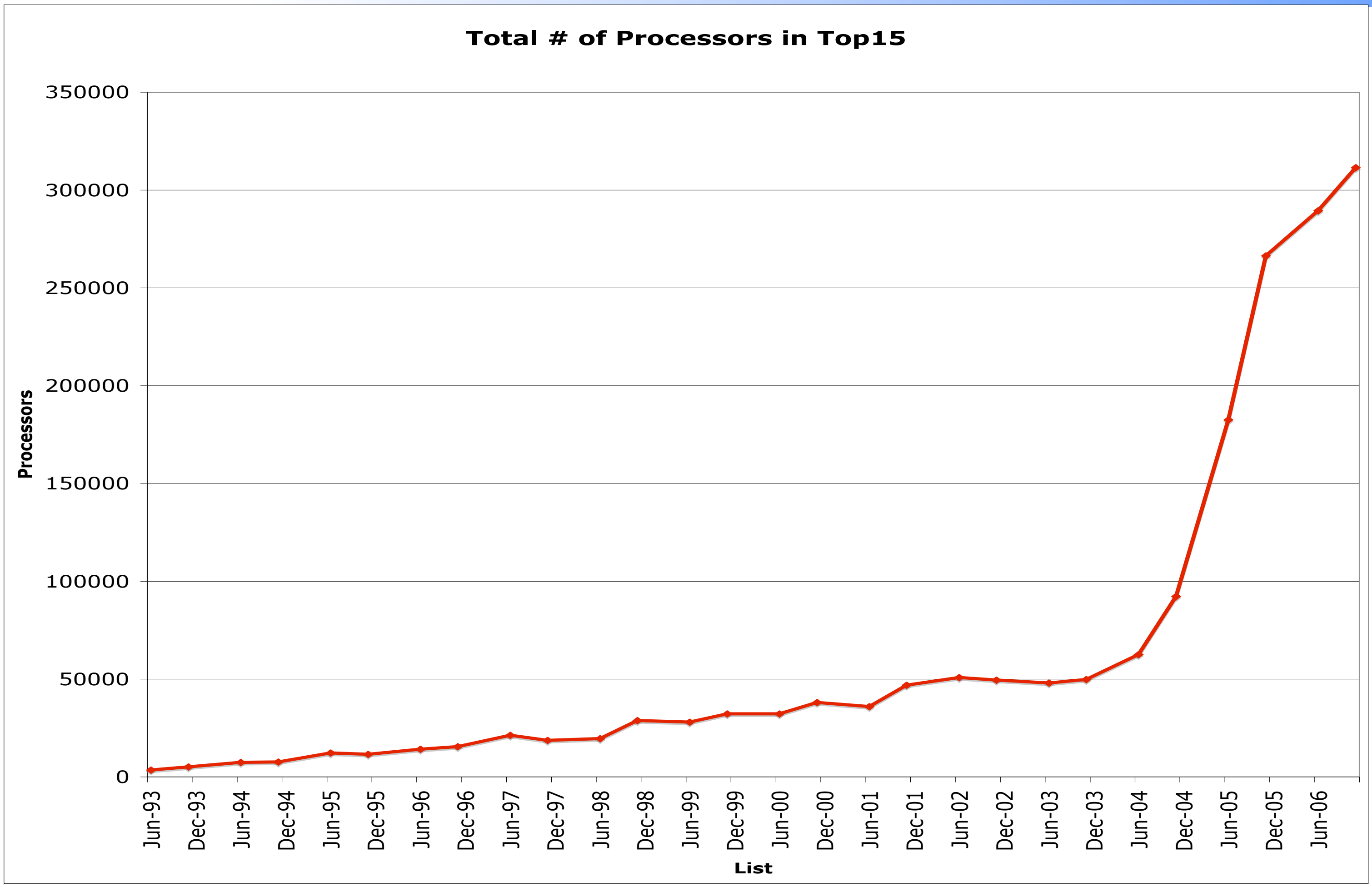
- Describe data layout in memory
  - Typically only have to do once after code startup
  - exception for adaptive codes, but there are not too many of them
- Describe desired layout on disk *or* desired logical layout
- Say “commit” when you want to write it out
- I/O subsystem requests data from compute nodes in optimal order for storage subsystem

FSP Storage Recommendations

- Need Common Structures for Data Exchange
  - Must be able to compare data between simulation and experiment
  - Must be able to compare data between different simulations
  - Must be able to use output from one set of codes as boundary conditions for a different set of codes
  - Must be able to share visualization and analysis tools & software infrastructure
- Implementation (CS issues)
  - separate data model from file format
  - Develop veneer interfaces (APIs) to simplify data access for physics codes
  - utilize modern database-like file storage approaches (hierarchical, self-describing file formats)
- Approach (management & funding)
  - must be developed through agreements/compromises within community (not imposed by CS on the physics community)
  - not one format (many depending on area of data sharing)
  - requires some level of sustained funding to maintain and document the data models & associated software infrastructure (data storage always evolves, just as the physics models and ITER engineering design evolves)

Comments about Performance for Multicore

The Future of HPC System Concurrency



Must ride exponential wave of increasing concurrency for forseeable future!
You will hit 1M cores sooner than you think!

Scalable I/O Issues For High On-Chip Concurrency

- **Scalable I/O for massively concurrent systems!**
  - Many issues with coordinating access to disk within node (on chip or CMP)
  - OS will need to devote more attention to QoS for cores competing for finite resource *(mutex locks and greedy resource allocation policies will not do!) (it is rugby where device == the ball)*

nTasks	I/O Rate 16 Tasks/node	I/O Rate 8 tasks per node
8	-	131 Mbytes/sec
16	7 Mbytes/sec	139 Mbytes/sec
32	11 Mbytes/sec	217 Mbytes/sec
64	11 Mbytes/sec	318 Mbytes/sec
128	25 Mbytes/sec	471 Mbytes/sec

Old OS Assumptions are Bogus on Hundreds of Cores

- **Assumes limited number of CPUs that must be shared**
  - *Old OS: time-multiplexing (context switching and cache pollution!)*
  - *New OS: spatial partitioning*
- **Greedy allocation of finite I/O device interfaces (eg. 100 cores go after the network interface simultaneously)**
  - *Old OS: First process to acquire lock gets device (resource/lock contention! Nondet delay!)*
  - *New OS: QoS management for symmetric device access*
- **Background task handling via threads and signals**
  - *Old OS: Interrupts and threads (time-multiplexing) (inefficient!)*
  - *New OS: side-cores dedicated to DMA and async I/O*
- **Fault Isolation**
  - *Old OS: CPU failure --> Kernel Panic (will happen with increasing frequency in future silicon!)*
  - *New OS: CPU failure --> Partition Restart (partitioned device drivers)*
- **Old OS invoked any interprocessor communication or scheduling vs. direct HW access**

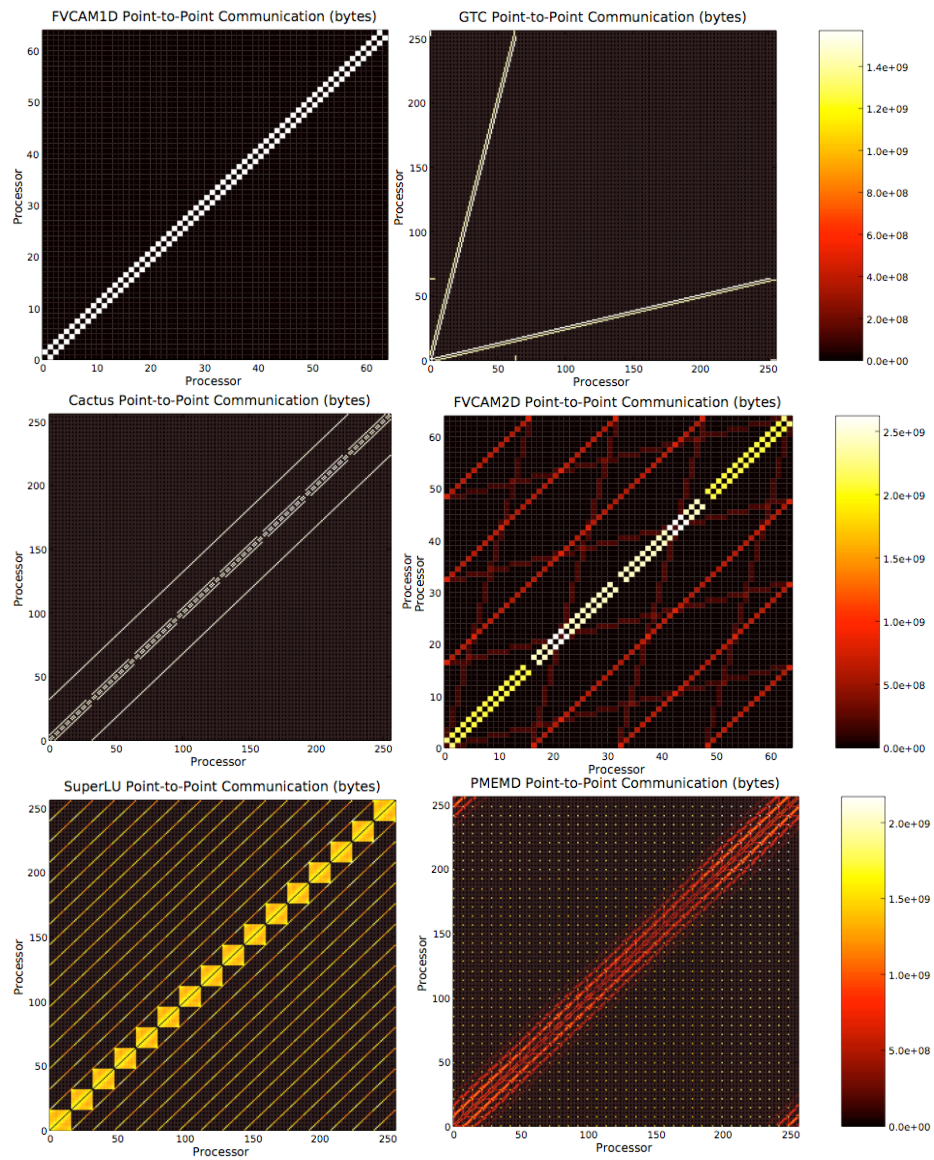
- ***New OS/CMP contract***
  - *No Time Multiplexing: Spatial partitioning*
  - *No interrupts: use side-cores*
  - *Resource Management: Need QoS policy enforcement at deepest level of chip and OS*

Comments about Interconnect Performance

NERSC

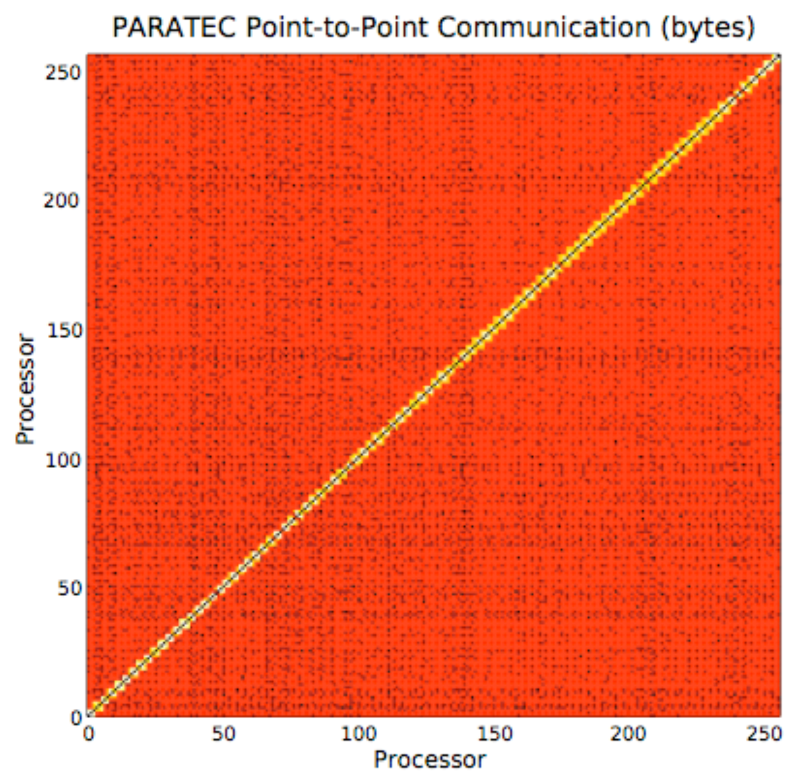
Interconnect Design Considerations for Massive Concurrency

- **Application studies provide insight to requirements for Interconnects (both on-chip and off-chip)**
  - On-chip interconnect is 2D planar (crossbar won't scale!)
  - Sparse connectivity for dwarfs; crossbar is overkill
  - No single best topology
- **A Bandwidth-oriented network for data**
  - Most point-to-point message exhibit sparse topology & bandwidth bound
- **Separate Latency-oriented network for collectives**
  - E.g., Thinking Machines CM-5, Cray T3D, IBM BlueGene/L&P
- **Ultimately, need to be aware of the on-chip interconnect topology in addition to the off-chip topology**
  - Adaptive topology interconnects (HFAST)
  - Intelligent task migration?



Interconnects
Need For High Bisection Bandwidth

- **3D FFT easy-to-identify as needing high bisection**
  - Each processor must send messages to all PE's! (all-to-all) for 1D decomposition
  - However, most implementations are currently limited by overhead of sending small messages!
  - 2D domain decomposition (required for high concurrency) actually requires sqrt(N) communicating partners! *(some-to-some)*
- **Same Deal for AMR**
  - AMR communication is sparse, but by no means is it bisection bandwidth limited

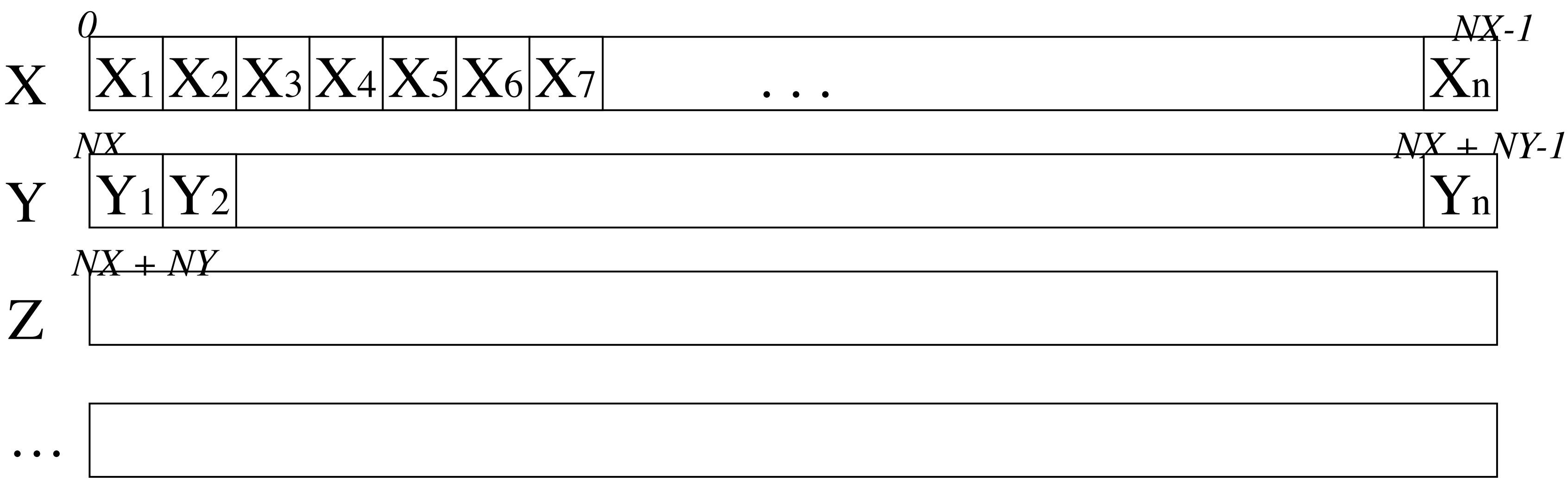


Accelerator Modeling Data

- Point data
  - Electrons or protons
  - Millions or billions in a simulation
  - Distribution is non-uniform
    - Fixed distribution at start of simulation
    - Change distribution (load balancing) each iteration
- Attributes of a point
  - Location: (double) x,y,z
  - Phase: (double) mx,my,mz
  - ID: (int64) id
  - Other attributes

Accelerator Modeling Data

Storage Format



Laid out sequentially on disk

Some formats are interleaved,

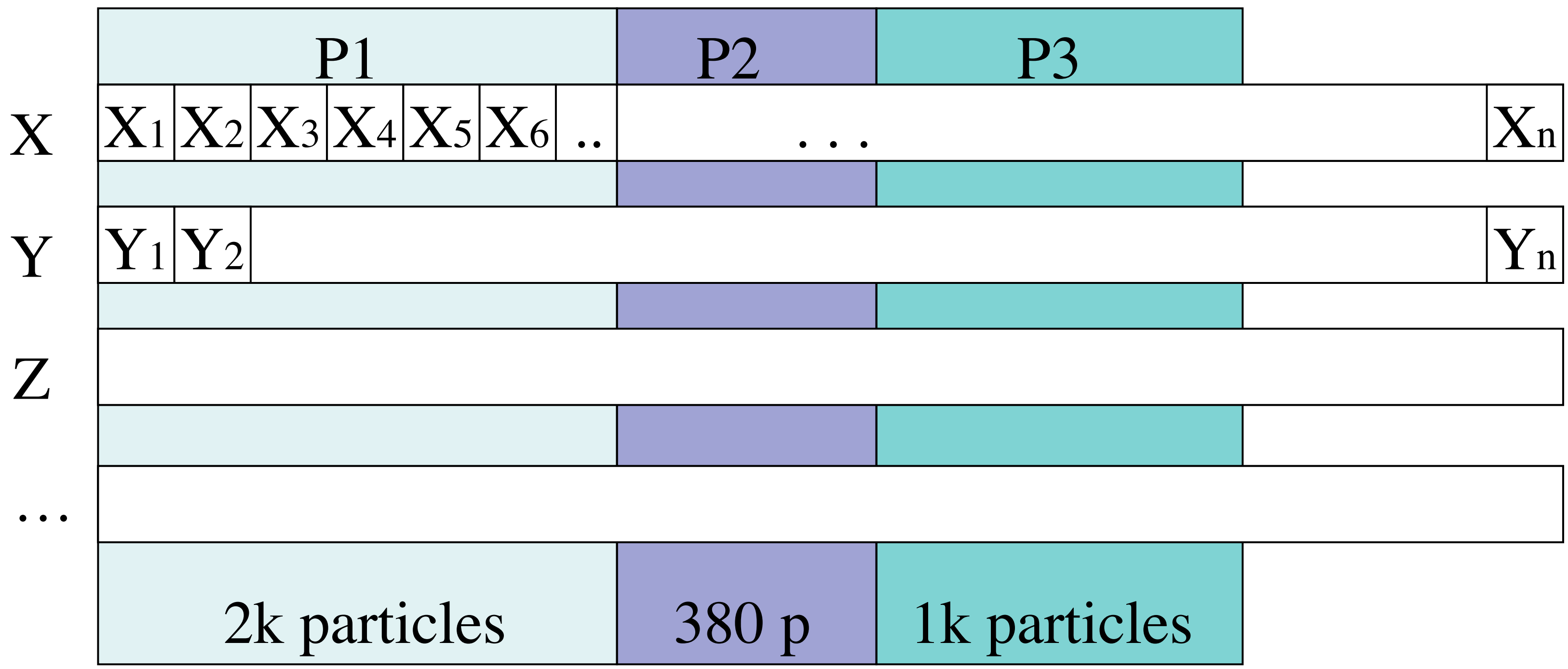
but causes problems for data analysis

Easier to reorganize in memory than on disk!

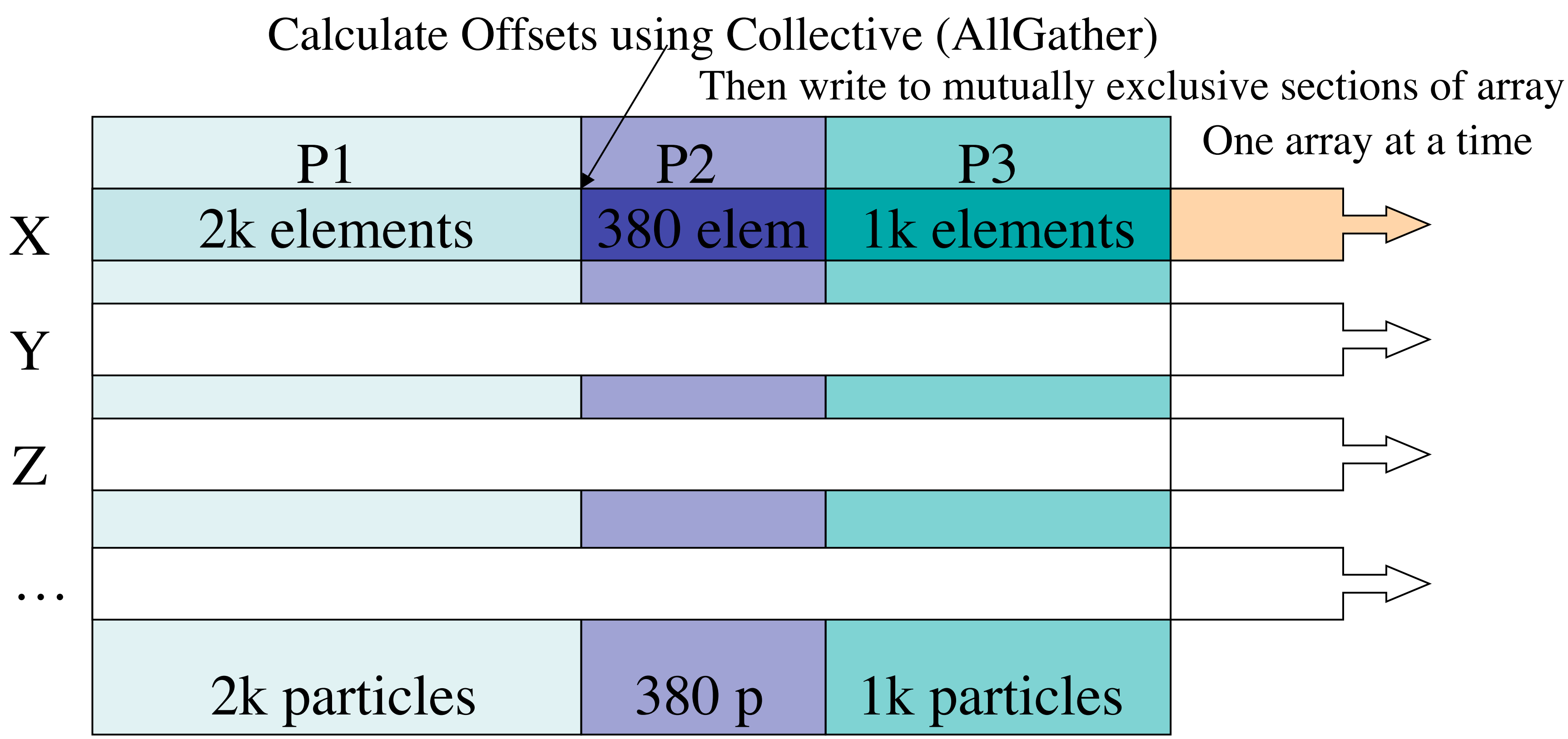
Office of Science
U.S. DEPARTMENT OF ENERGY

Accelerator Modeling Data

Storage Format



Accelerator Modeling Data



Still suffers from alignment issues…

Accelerator Modeling Benchmark

Mode	Global Performance	Per Processor Performance
One file per processor Raw binary	1288 MB/s	20 MB/s
Parallel I/O (1-file) Raw binary MPI-IO	241 MB/s	3 MB/s
Parallel I/O (1-file) pHDF5 -- H5Part	773 MB/s	12 MB/s

Seaborg: 64nodes,
1024 processors,
780 Gbytes of data total

Office of Science
U.S. DEPARTMENT OF ENERGY

Physical Layout Tends to Result in Handful of I/O Patterns

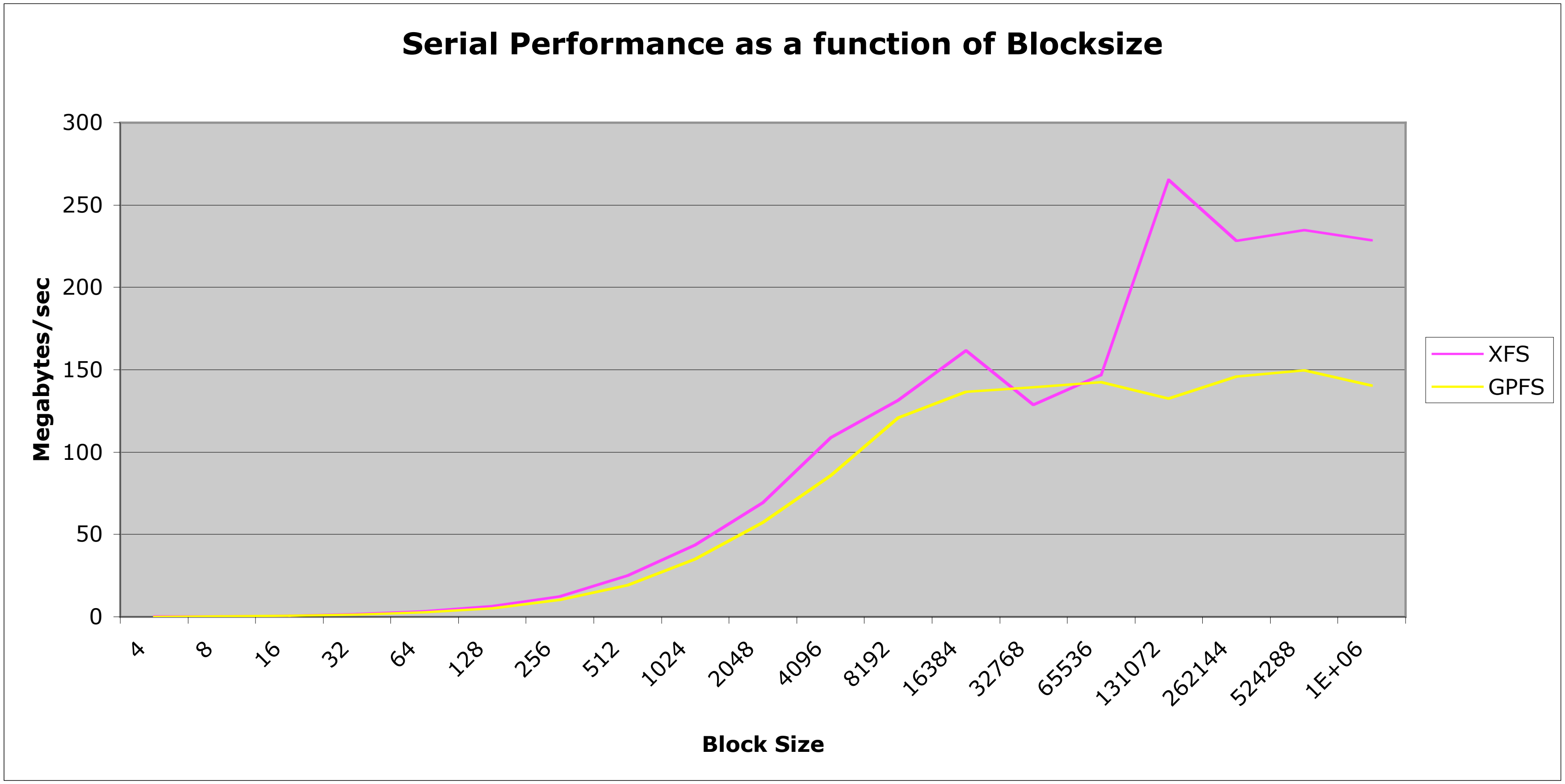
- **2D-3D I/O patterns (striding)**
  - **1 file per processor** *(Raw Binary and HDF5)*
    - *Raw binary assesses peak performance*
    - *HDF5 determines overhead of metadata, data encoding, and small accesses associated with storage of indices and metadata*
  - **1-file reverse domain decomp** *(Raw MPI-IO and pHDF5)*
    - *MPI-IO is baseline (peak performance)*
    - *Assess pHDF5 or pNetCDF implementation overhead*
  - **1-file chunked** *(Raw MPI-IO and pHDF5)*
- **1D I/O patterns (writing to distinct 1D offsets)**
  - *Same as above, but for 1D data layouts*
  - *1-file per processor is same in both cases*
- ***MadBench?***
  - *Out-of-Core performance (emphasizes local filesystem?)*

GPFS MPI-I/O Experiences

nTasks	I/O Rate 16 Tasks/node	I/O Rate 8 tasks per node
8	-	131 Mbytes/sec
16	7 Mbytes/sec	139 Mbytes/sec
32	11 Mbytes/sec	217 Mbytes/sec
64	11 Mbytes/sec	318 Mbytes/sec
128	25 Mbytes/sec	471 Mbytes/sec

- Block domain decomp of 512^3 3D 8-byte/element array in memory written to disk as single undecomposed 512^3 logical array.
- Average throughput for 5 minutes of writes x 3 trials
- Issue is related to LAPI lock contention…

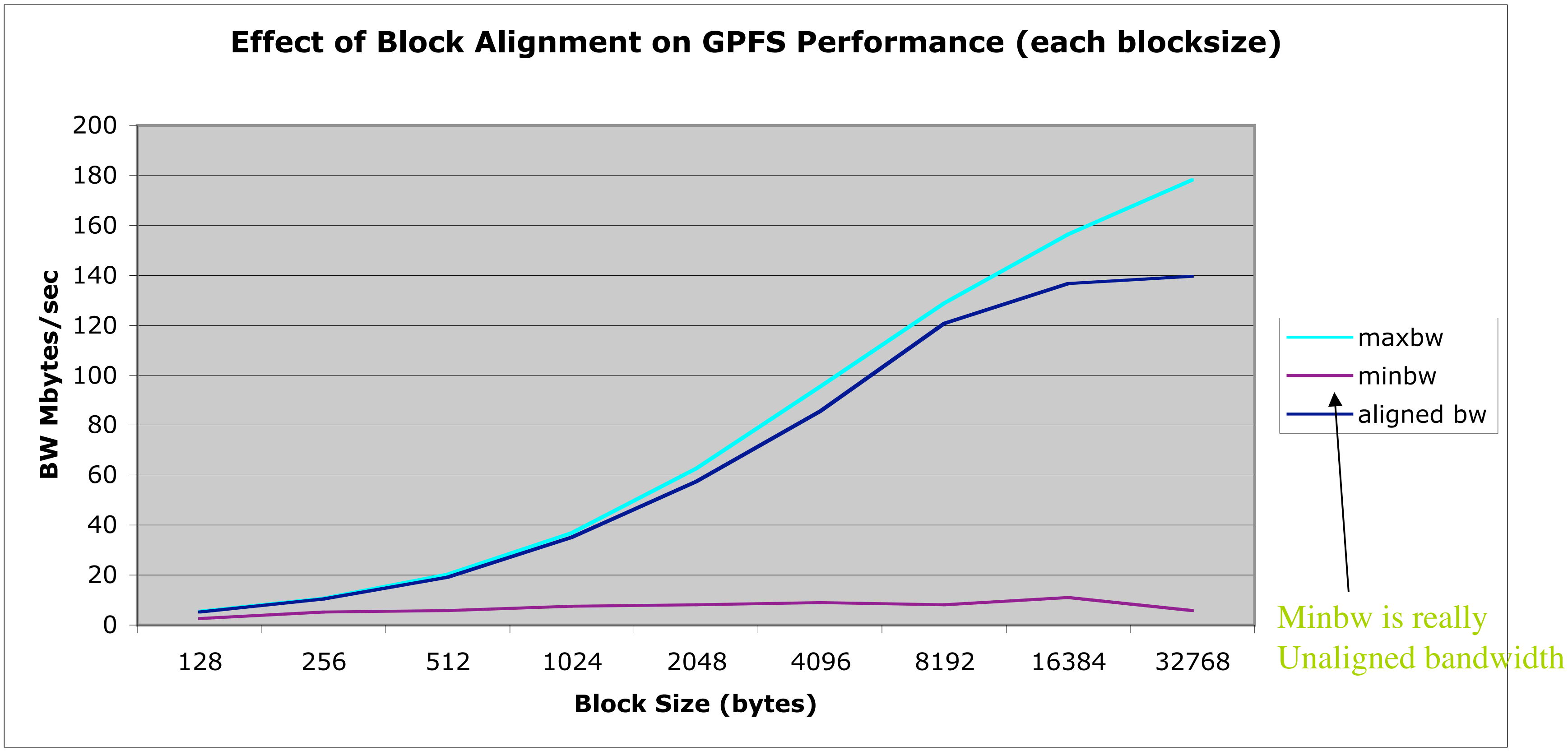
GPFS: BW as function of write length



Block Aligned on disk!
Page Aligned in memory!

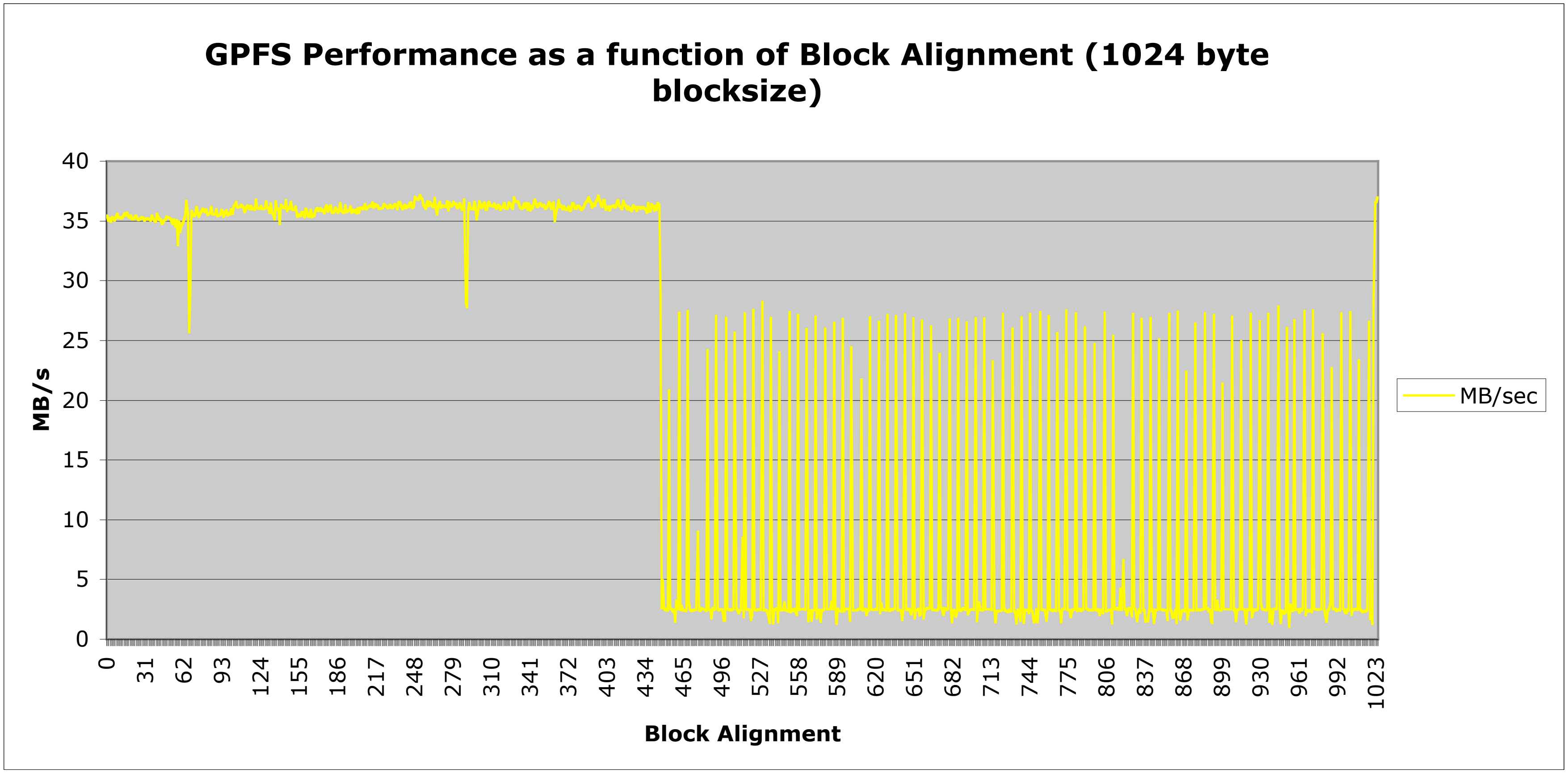
Amdahl's law effects for Metadata storage…

GPFS (unaligned accesses)

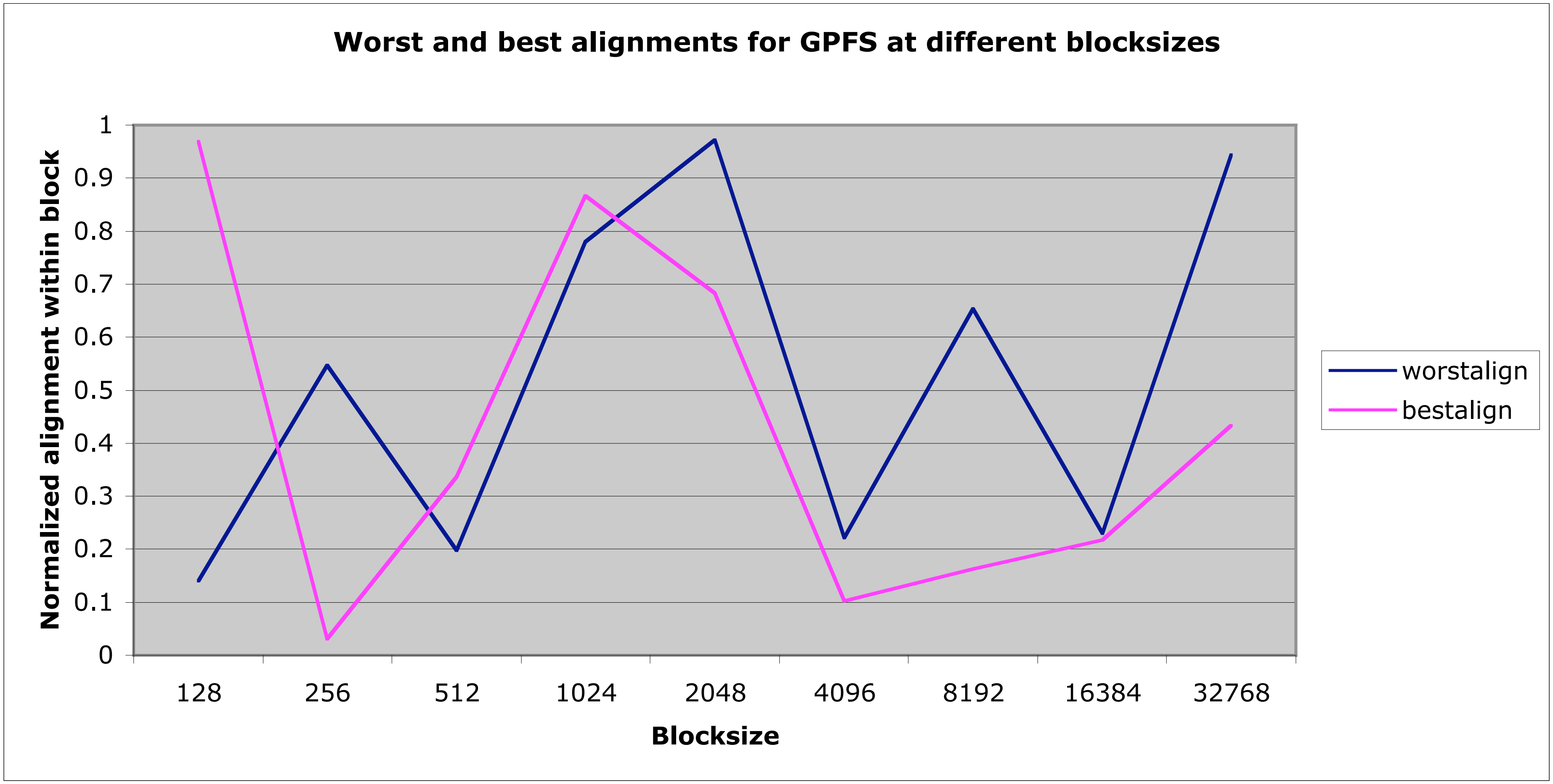


Unaligned access sucks!

GPFS: Unaligned accesses



GPFS: (what alignment is best?)



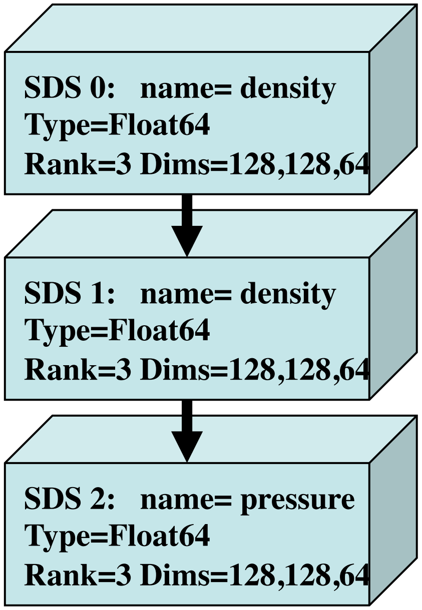
No consistently "best" alignment except for perfect block alignment!

That means 256k block boundaries for GPFS!

Higher-Level Storage Organization

HDF4/NetCDF Data Model

- **Datasets**
  - Name
  - Datatype
  - Rank,Dims

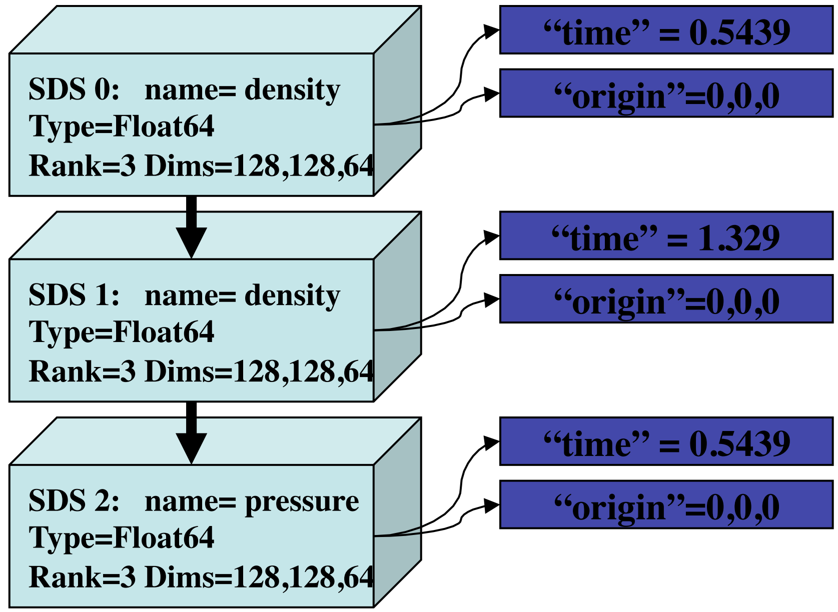


Datasets are inserted sequentially to the file

Can be randomly accessed on read

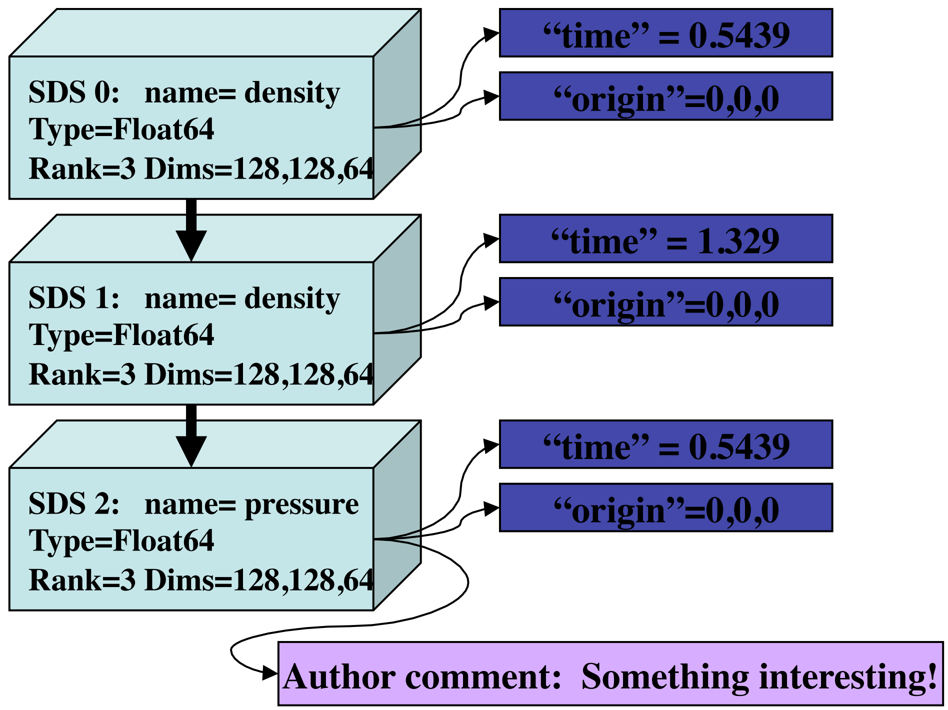
HDF4/NetCDF Data Model

- **Datasets**
  - Name
  - Datatype
  - Rank,Dims
- **Attributes**
  - Key/value pair
  - DataType and length



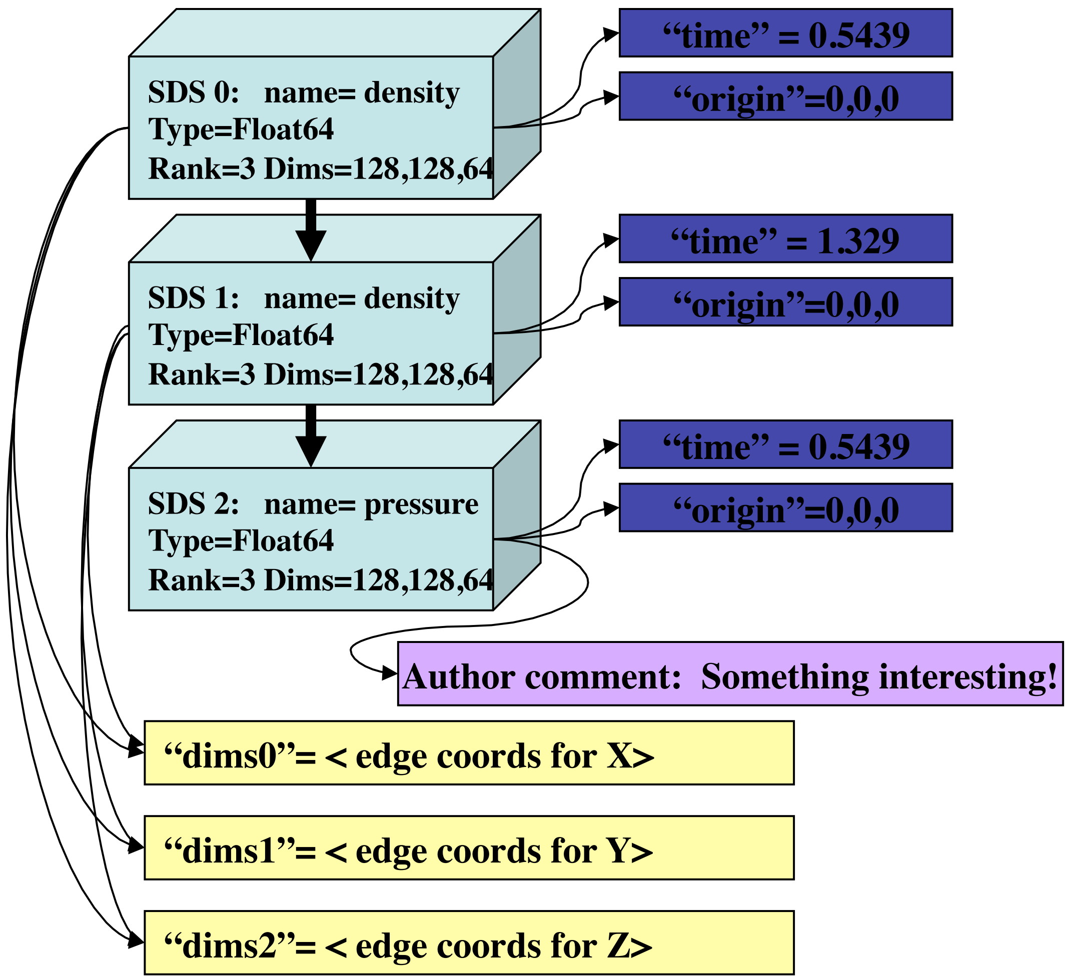
HDF4/NetCDF Data Model

- Datasets
  - Name
  - Datatype
  - Rank,Dims
- Attributes
  - Key/value pair
  - DataType and length
- Annotation
  - Freeform text
  - String Termination



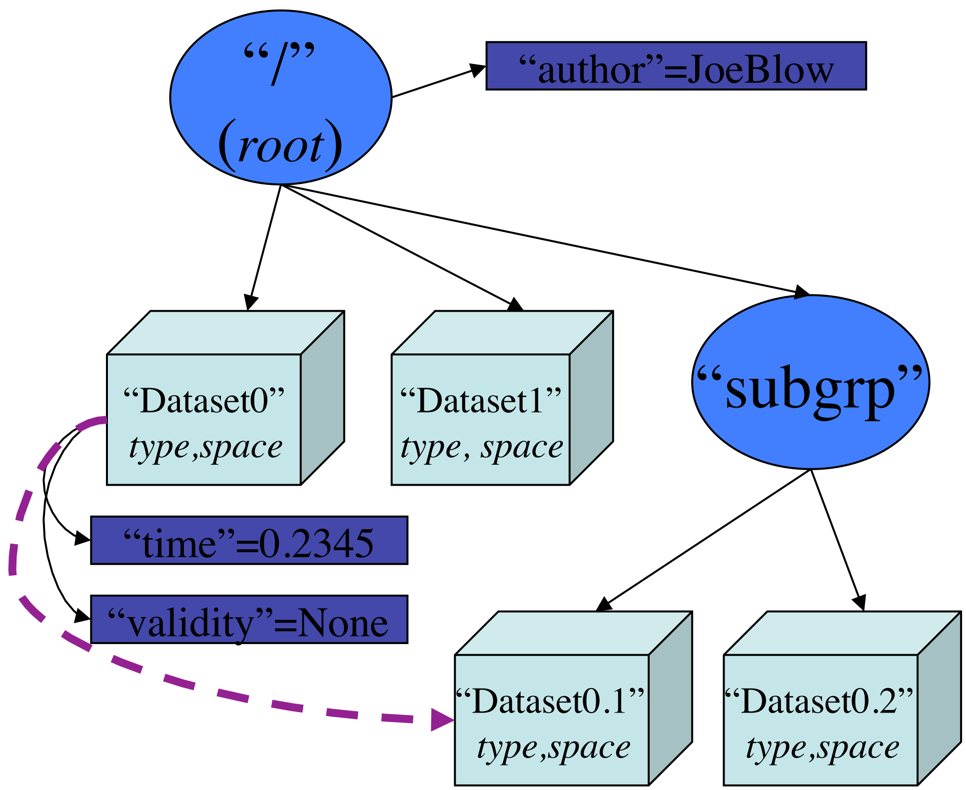
HDF4/NetCDF Data Model

- **Datasets**
  - Name
  - Datatype
  - Rank,Dims
- **Attributes**
  - Key/value pair
  - DataType and length
- **Annotation**
  - Freeform text
  - String Termination
- **Dimensions**
  - Edge coordinates
  - Shared attribute



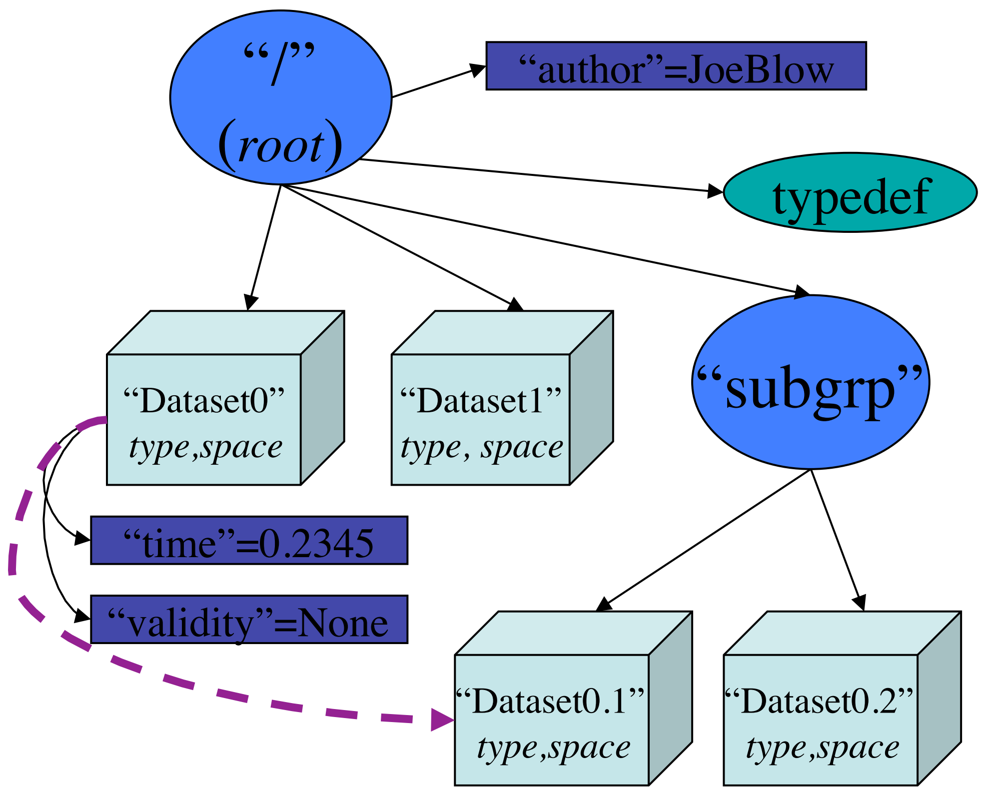
HDF5 Data Model

- Groups
  - Arranged in directory hierarchy
  - root group is always '/'
- Datasets
  - Dataspace
  - Datatype
- Attributes
  - Bind to Group & Dataset
- References
  - Similar to softlinks
  - Can also be subsets of data



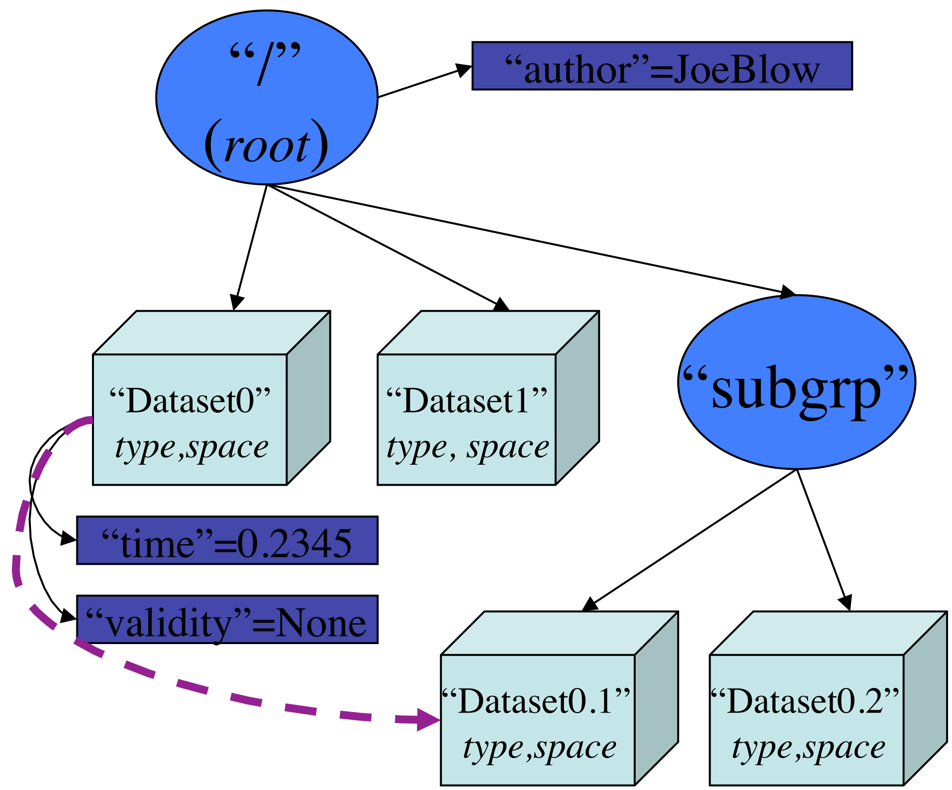
HDF5 Data Model (funky stuff)

- Complex Type Definitions
  - Not commonly used feature of the data model.
  - Potential pitfall if you commit complex datatypes to your file
- Comments
  - Yes, annotations actually do live on.



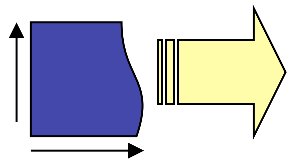
HDF5 Data Model (caveats)

- Flexible/Simple Data Model
  - You can do anything you want with it!
  - You typically define a higher level data model on top of HDF5 to describe domain-specific data relationships
  - Trivial to represent as XML!
- The perils of flexibility!
  - Must develop community agreement on these data models to *share* data effectively
  - Multi-Community-standard data models required across for reusable visualization tools
  - Preliminary work on Images and tables

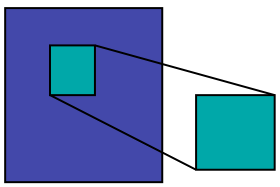


Data Storage Layout / Selections

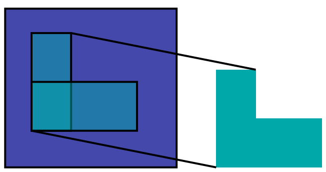
- Elastic Arrays



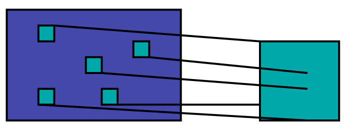
- Hyperslabs
  - Logically contiguous chunks of data
  - Multidimensional Subvolumes
  - Subsampling (striding, blocking)



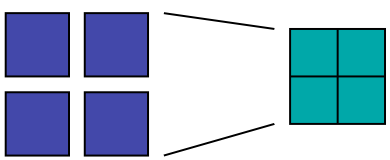
- Union of Hyperslabs
  - Reading a non-rectangular sections



- Gather/Scatter



- Chunking
  - Usually for efficient Parallel I/O



Dataspace Selections (H5S)

Transfer a subset of data from disk to fill a memory buffer

Disk Dataspace

H5Sselect_hyperslab(disk_space, H5S_SELECT_SET,
offset[3]={1,2},NULL,count[2]={4,6},NULL)

Memory Dataspace

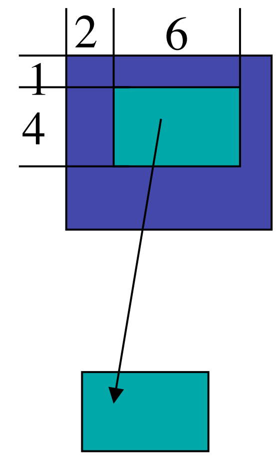
mem_space = H5S_ALL

Or

mem_space =H5Dcreate(rank=2,dims[2]={4,6});

Transfer/Read operation

H5Dread(dataset,mem_datatype, mem_space, disk_space,
H5P_DEFAULT, mem_buffer);



Dataspace Selections (H5S)

Transfer a subset of data from disk to subset in memory

Disk Dataspace

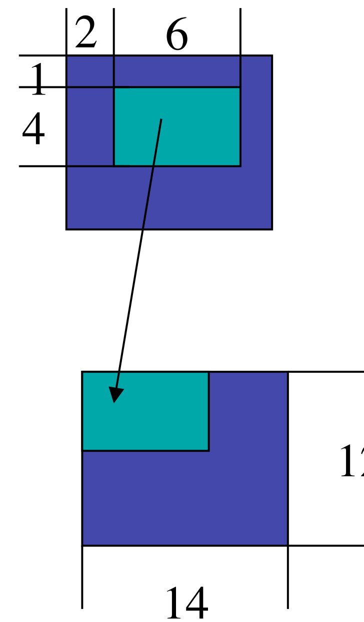
H5Sselect_hyperslab(disk_space, H5S_SELECT_SET,
offset[3]={1,2},NULL,count[2]={4,6},NULL)

Memory Dataspace

mem_space =H5Dcreate_simple(rank=2,dims[2]={12,14});
H5Sselect_hyperslab(mem_space, H5S_SELECT_SET,
offset[3]={0,0},NULL,count[2]={4,6},NULL)

Transfer/Read operation

H5Dread(dataset,mem_datatype, mem_space, disk_space,
H5P_DEFAULT, mem_buffer);

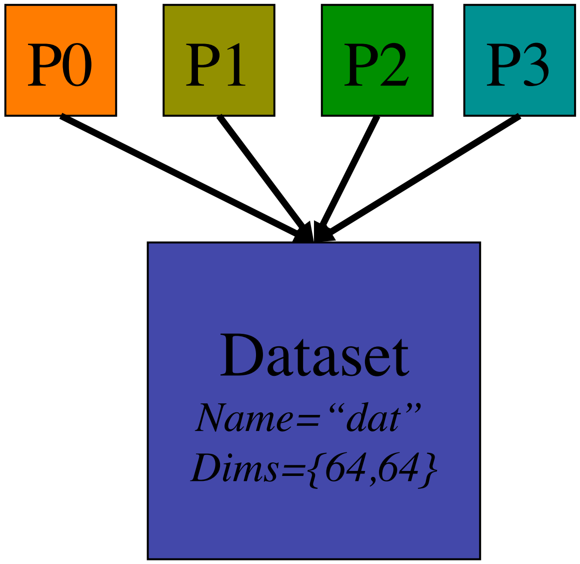


pHDF5 (example 1)

•File open requires explicit selection of Parallel I/O layer.
•All PE's collectively open file and declare the overall size of the dataset.

All MPI Procs!

```
props = H5Pcreate(H5P_FILE_ACCESS);
/* Create file property list and set for Parallel I/O */
H5Pset_fapl_mpio(prop, MPI_COMM_WORLD,
            MPI_INFO_NULL);
file=H5Fcreate(filename,H5F_ACC_TRUNC,
            H5P_DEFAULT,props); /* create file */
H5Pclose(props); /* release the file properties list */
filespace = H5Screate_simple(rank=2,dims[2]={64,64},
            NULL)
dataset = H5Dcreate(file,"dat",H5T_NATIVE_INT,
            space,H5P_DEFAULT); /* declare dataset */
```

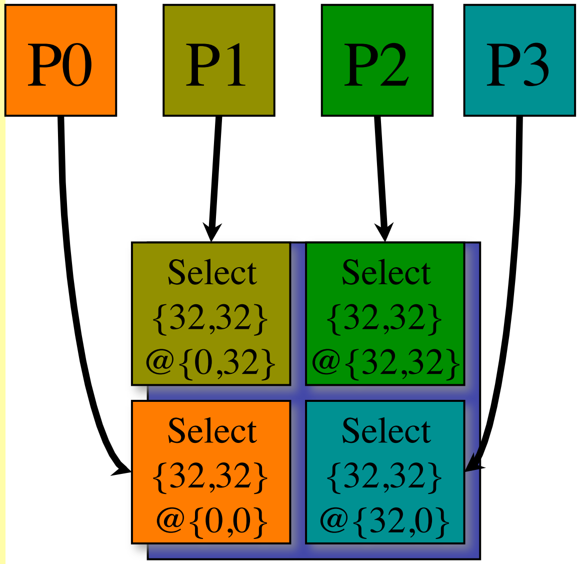


pHDF5 (example 1 cont…)

- Each proc selects a hyperslab of the dataset that represents its portion of the domain-decomposed dataset and read/write collectively or independently.

All MPI Procs!

```
/* select portion of file to write to */
H5Sselect_hyperslab(filespace, H5S_SELECT_SET,
        start= P0{0,0}:P1{0,32}:P2{32,32}:P3{32,0},
        stride= {32,1},count={32,32},NULL);
/* each proc independently creates its memspace */
memspace = H5Screate_simple(rank=2,dims={32,32},
        NULL);
/* setup collective I/O prop list */
xfer_plist = H5Pcreate (H5P_DATASET_XFER);
H5Pset_dxpl_mpio(xfer_plist, H5FD_MPIO_COLLECTIVE);
H5Dwrite(dataset,H5T_NATIVE_INT, memspace, filespace,
        xfer_plist, local_data); /* write collectively */
```



Serial I/O Benchmarks

System	HDF4.1r5 (netCDF)	HDF5 v1.4.4	FlexIO (Custom)	F77 Unf
SGI Origin 3400 ***(escher.nersc.gov)***	**111M/s**	**189M/s**	**180M/s**	**140M/s**
IBM SP2 ***(seaborg.nersc.gov)***	**65M/s**	**127M/s**	**110M/s**	**110M/s**
Linux IA32 ***(platinum.ncsa.uiuc.edu)***	**34M/s**	**40M/s**	**62M/s**	**47M/s**
Linux IA64 Teragrid node ***(titan.ncsa.uiuc.edu)***	**26M/s**	**83M/s**	**77M/s**	**112M/s**
NEC/Cray SX-6 ***(rime.cray.com)***				

- Write 5-40 datasets of 128^3 DP float data
- Single CPU (multiple CPU's can improve perf. until interface saturates)
- Average of 5 trials

GPFS MPI-I/O Experiences

nTasks	I/O Rate 16 Tasks/node	I/O Rate 8 tasks per node
8	-	131 Mbytes/sec
16	7 Mbytes/sec	139 Mbytes/sec
32	11 Mbytes/sec	217 Mbytes/sec
64	11 Mbytes/sec	318 Mbytes/sec
128	25 Mbytes/sec	471 Mbytes/sec

- Block domain decomp of 512^3 3D 8-byte/element array in memory written to disk as single un-decomposed 512^3 logical array.
- Average throughput for 5 minutes of writes x 3 trials